

# Lecture Notes in Computer Science

1982

**Stefan Näher Dorothea Wagner (Eds.)**

## Algorithm Engineering

**4th International Workshop, WAE 2000  
Saarbrücken, Germany, September 2000  
Proceedings**



**Springer**

26  
26  
26

Springer

*B*

*H*

*N*

*S*

*HK*

*L*

*M*

*P*

*U*

BN h BW

HE

HW  
SD  
P



Springer

82

8281  
8282  
8283

82

8N h

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

158

8N

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

8281

## Preface

This volume contains the papers accepted for the 4th *Workshop on Algorithm Engineering* (WAE 2000) held in Saarbrücken, Germany, during 5–8 September 2000, together with the abstract of the invited lecture given by Karsten Weihe. The Workshop on Algorithm Engineering covers research on all aspects of the subject. The goal is to present recent research results and to identify and explore directions for future research. Previous meetings were held in Venice (1997), Saarbrücken (1998), and London (1999).

Papers were solicited describing original research in all aspects of algorithm engineering, including:

- Development of software repositories and platforms which allow the use of and experimentation with efficient discrete algorithms.
- Novel uses of discrete algorithms in other disciplines and the evaluation of algorithms for realistic environments.
- Methodological issues including standards in the context of empirical research on algorithms and data structures.
- Methodological issues regarding the process of converting user requirements into efficient algorithmic solutions and implementations.

The program committee accepted 16 from a total of 30 submissions. The program committee meeting was conducted electronically. The criteria for selection were originality, quality, and relevance to the subject area of the workshop. Considerable effort was devoted to the evaluation of the submissions and to providing the authors with feedback. Each submission was reviewed by at least four program committee members (assisted by subreferees). A special issue of the *ACM Journal of Experimental Algorithmics* will be devoted to selected papers from WAE 2000.

We would like to thank all those who submitted papers for consideration, as well as the program committee members and their referees for their contributions. We gratefully acknowledge the dedicated work of the organizing committee, and the help of many volunteers. We thank all of them for their time and effort.

July 2001

Stefan Näher  
Dorothea Wagner

## Invited Lecturer

*Karsten Weihe*

Institut für Diskrete Mathematik, Bonn

## Program Committee

*Michael Goodrich*

John Hopkins University

*Dan Halperin*

Tel Aviv University

*Mike Jünger*

Universität zu Köln

*Thomas Lengauer*

GMD Bonn

*Joe Marks*

MERL, Cambridge

*Stefan Näher*, Co-chair

Universität Trier

*Mark Overmars*

Universiteit Utrecht

*Steven Skiena*

State University of NY, Stony Brook

*Jack Snoeyink*

University of North Carolina, Chapel Hill

*Roberto Tamassia*

Brown University

*Dorothea Wagner*, Co-chair

Universität Konstanz

*Peter Widmayer*

ETH Zürich

## Organizing Committee

*Uwe Brahm*

MPI Saarbrücken

*Hop Sibeyn*, Chair

MPI Saarbrücken

*Christoph Storb*

MPI Saarbrücken

*Roxane Wetzel*

MPI Saarbrücken

## Referees

Arne Andersson, Lars A. Arge, Stephen Aylward, Sanjoy Baruah, Mark de Berg, Hans L. Bodlaender, Matthias Buchheim, Adam Buchsbaum, Irit Dinur, Shlomo Dubnov, Matthias Elf, Martin Farach-Colton, Uriel Feige, Andrew Goldberg, Leslie Hall, Eran Halperin, Sarel Har-Peled, Han Hoogeveen, Kevin Jeffay, Lutz Kettner, Vladlen Koltun, Marc van Kreveld, Joachim Kupke, Olivier Lartillot, Frauke Liers, Bernard Moret, Michal Ozery, Oded Regev, A.F. van der Stappen, Cliff Stein, Sivan Toledo, Marinus Veldhorst, Remco Velthkamp, Bram Verweij, Karsten Weihe, Thomas Willhalm, Martin Wolff

## Table of Contents

### Invited Lectures

On the Differences between “Practical” and “Applied” . . . . .	1
<i>Karsten Weihe</i>	

### Contributed Papers

An Experimental Study of Online Scheduling Algorithms . . . . .	11
<i>Susanne Albers and Bianca Schröder</i>	
Implementation of $O(nm \log n)$ Weighted Matchings in General Graphs. The Power of Data Structures . . . . .	23
<i>Kurt Mehlhorn and Guido Schäfer</i>	
Pushing the Limits in Sequential Sorting . . . . .	39
<i>Stefan Edelkamp and Patrick Stiegeler</i>	
Efficient Sorting Using Registers and Caches . . . . .	51
<i>Lars Arge, Jeff Chase, Jeffrey S. Vitter, and Rajiv Wickremesinghe</i>	
Lattice Basis Reduction with Dynamic Approximation . . . . .	63
<i>Werner Backes and Susanne Wetzel</i>	
Clustering Data without Prior Knowledge . . . . .	74
<i>Javed Aslam, Alain Leblanc, and Clifford Stein</i>	
Recognizing Bundles in Time Table Graphs – A Structural Approach . . . . .	87
<i>Annegret Liebers and Karsten Weihe</i>	
Analysis and Experimental Evaluation of an Innovative and Efficient Routing Protocol for Ad-hoc Mobile Networks . . . . .	99
<i>I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis</i>	
Portable List Ranking: An Experimental Study . . . . .	111
<i>Isabelle Guérin Lassous and Jens Gustedt</i>	
Parallelizing Local Search for CNF Satisfiability Using Vectorization and PVM . . . . .	123
<i>Kazuo Iwama, Daisuke Kawai, Shuichi Miyazaki, Yasuo Okabe, and Jun Umemoto</i>	
Asymptotic Complexity from Experiments? A Case Study for Randomized Algorithms . . . . .	135
<i>Peter Sanders and Rudolf Fleischer</i>	



VIII Table of Contents

Visualizing Algorithms over the Web with the Publication-Driven Approach . . . . .	147
<i>Camil Demetrescu, Irene Finocchi, and Giuseppe Liotta</i>	
Interchanging Two Segments of an Array in a Hierarchical Memory System . . . . .	159
<i>Jesper Bojesen and Jyrki Katajainen</i>	
Two-Dimensional Arrangements in CGAL and Adaptive Point Location for Parametric Curves . . . . .	171
<i>Iddo Hanniel and Dan Halperin</i>	
Planar Point Location for Large Data Sets: To Seek or Not to Seek . . . . .	183
<i>Jan Vahrenhold and Klaus H. Hinrichs</i>	
Implementation of Approximation Algorithms for Weighted and Unweighted Edge-Disjoint Paths in Bidirected Trees . . . .	195
<i>Thomas Erlebach and Klaus Jansen</i>	
Dynamic Maintenance Versus Swapping: An Experimental Study on Shortest Paths Trees . . . . .	207
<i>Guido Proietti</i>	
Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study . . . . .	218
<i>Camil Demetrescu, Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni</i>	
New Algorithms for Examination Timetabling . . . . .	230
<i>Massimiliano Caramia, Paolo Dell’Olmo, and Giuseppe F. Italiano</i>	
<b>Author Index</b> . . . . .	243

# On the Differences between “Practical” and “Applied”

Karsten Weihe

Forschungsinstitut für Diskrete Mathematik  
Lennéstr. 2, 53113 Bonn, Germany  
[weihek@acm.org](mailto:weihek@acm.org)

The terms “practical” and “applied” are often used synonymously in our community. For the purpose of this talk I will assign more precise, distinct meanings to both terms (which are not intended to be ultimate definitions). More specifically, I will reserve the word “applied” for work

whose crucial, central goal is finding a feasible, reasonable (*e.g.* economical) solution to a concrete real-world problem, which is requested by someone outside theoretical computer science for his or her own work.

In contrast, “practical” then will refer to every other sort of implementation-oriented work. Most of the work published so far in WAE proceedings, AL(EN)EX proceedings, and “applied” tracks of other conferences in theoretical computer science is practical, not applied, in this spirit.

Many people got the fundamental experience that applied work is different and obeys its own rules. Since both practical and applied work is very versatile, it is hard to catch the exact differences. To get to the point, I will reduce the versatility of practical work to a “pattern” of sound practical work, which (to my feeling) reflects the current practice in our community quite well. The discussion of the differences between “practical” and “applied” is then broken down into discussions of the individual items in this pattern, which are more focused and hopefully more substantiated.

The methodology of the talk is to formulate one (or more) differences between “practical” and “applied” for each item and to substantiate them by “war stories” (in the sense of Skiena [13]). For brevity, most details (and all figures) are omitted. Selected aspects are discussed in greater detail in [15].

## Practical vs. Applied

In this talk and abstract, the difference between “practical” and “applied” is roughly identical to the difference between natural sciences and engineering sciences:

- In natural sciences, the “dirty” details of reality are abstracted away, and a simplified, coherent scenario is analysed under “lab conditions.”
- Engineering sciences aim at a feasible, reasonable (*e.g.* economical) solution to the use case at hand.

For example, the restriction to numerical criteria such as the run time and the space consumption is already an abstraction; the effort for implementing an algorithm and maintaining the implementation is often much more important and must be taken into account in engineering work.

## “Pattern” for Sound Practical Work

So far, there is no standardized pattern of sound practical work, although there are various attempts to specify rules how to perform sound practical studies [2,3,5,6,7,8,9,10]. However, it seems to me that the following preliminary, immature pattern comes close to the de-facto standard established by the majority of practical papers specifically from our community:

- Choose an exact, clean, well-formalized problem.
- Implement algorithms from the theoretical literature and (optionally) own new variants/algorithms.
- Collect input instances from random generators and (optionally) public benchmark sets.
- Run the algorithms on these instances and collect statistical data on CPU time, operation counts, etc.
- Evaluate the statistics to compare the algorithms.

Here is my personal list of significant differences:

- Choose an exact, clean, well-formalized problem...  
... but the problem may be highly underspecified and volatile.
- Implement algorithms from the (theoretical) literature...  
... but the algorithms from the literature may be inappropriate.
- Collect input instances from random generators...  
... but random instances may be very different from real-world instances and thus mislead the research.
- Make statistics on CPU time, operation counts, etc...  
... but what to measure is not always clear.  
... but non-quantifiable characteristics are often more important.
- Compare the algorithms...  
... but the algorithms may solve different specifications of the underspecified problem.

## Detailed Differences

### Difference 1:

- Choose an exact, clean, well-formalized problem...  
... but the problem may be highly underspecified and volatile.

Very often, crucial details of the problem definition are not understood even if the problem is purely mathematical in nature. Even worse, the problem may involve psychological, sociological, political, etc., aspects, which might hardly be expressible in mathematical terms.<sup>1</sup>

If we can find a mathematical model whose theoretical outcome comes close to the empirical observations (and is efficiently computable), fine. However, personally, I was mainly confronted with applications from the real world in which such a model was beyond our analytical skills. An insightful example was presented in [16,17]: the surface of a CAD workpiece is given as a *mesh*, that is, as a set of elementary surface patches (think of continuously deformed, curved polygons in the three-dimensional space). If two patches are intended to be neighbored on the approximated surface, they do not necessarily meet but are placed (more or less) close to each other. The problem is to reconstruct the neighborhood relations.

The methodological problem is this: the neighborhood relations are a purely subjective ingredient, namely the intention of the designer of the workpiece. In [16] we presented a computational study, which suggests that the data is too dirty to allow promising ad-hoc formalizations of the problem.<sup>2</sup> On the other hand, there is no evidence that a more sophisticated problem definition comes reasonably close to reality.

For a better understanding it might be insightful to compare this problem with another, more common class of problems (which stands for many others): *graph drawing*. The problem addressed in Difference 1 is certainly not new. Graph drawing is also an example for subjective ingredients, because the ultimate goal is not mathematical but aesthetical or cognitive. Nonetheless, various adequate mathematical models have been used successfully for real-world problems in this realm.

However, there is an important difference between these two real-world problems, and this difference is maybe insightful beyond this contrasting pair of specific examples. In the CAD problem from [16,17], there is exactly one correct solution. All human beings will identify it by a brief visual inspection of a picture of the whole workpiece, because the overall shape of a typical workpiece is easily analysed by the human cognitive apparatus. In other words, everybody recognizes the correct neighborhood relations intuitively,<sup>3</sup> but nobody can specify the rules that led him/her to the solution. In particular, every deviation from this solution could be mercilessly identified by a punctual human inspection.

Thus, although the problem definition is fuzzy, the evaluation of a solution by the end-user is potentially rigorous. In contrast, the aesthetical evaluation of

<sup>1</sup> L. Zadeh, the inventor of *fuzzy logic*, found a pregnant formulation of Difference 1: precision is the enemy of relevance (citation translated back from German and thus possibly not verbatim).

<sup>2</sup> For instance, one ad-hoc approach, which is (to our knowledge) the standard approach outside academia, is to regard two patches as neighbored if their “distance” according to some distance measure is smaller than a fixed threshold value.

<sup>3</sup> Except for rare pathological cases, in which it was hard or even impossible to guess the designer’s intention.

a graph drawing is as fuzzy as the problem definition itself. Roughly speaking, it suffices to “please” the end-user. To rephrase this difference a bit more provocatively: the strength of the meaning of “successful” may vary from application to application.

### Difference 2:

Implement algorithms from the (theoretical) literature...  
... but the algorithms from the literature may be inappropriate.

For example, a lot of work on various special cases of the general *scheduling* problem has been published throughout the last decades. Most theoretical work concentrates on polynomial special cases and makes extensive use of the restriction to a hand-picked collection of side constraints and objectives.

However, a typical real-world scheduling problem might involve a variety of side constraints and objectives. At least for me, there is no evidence (not to mention a concrete perspective) that any of these theoretical algorithms can be generalized to complex problem versions such that it truly competes with meta-heuristics like genetic algorithms and simulated annealing.

### Difference 3:

Collect input instances from random generators...  
... but random instances may be very different from real-world instances and thus mislead the research.

From the above example: what in the world is a random CAD workpiece?

Of course, we could try to identify certain “typical” statistical characteristics of CAD workpieces and then design a random generator whose outputs also have these characteristics. However, the profit from such an effort is not clear. If the “realistic” random instances reveal the same performance profile as the real-world instances, they do not give any additional information. On the other hand, if they provide a significantly different profile, does that tell us something about the expected performance of our algorithms in the application – or about the discrepancy between the real-world instances and our randomized approximation of reality?

Another result [14] may serve as an extreme, and thus extremely clarifying, example: for a given set of trains in some railroad network, the problem is to find a set of stations of minimal cardinality such that every train stops at one (or more) of them. We can regard each train as a set of stations, so the problem amounts to an application of the *hitting-set problem* (which is  $\mathcal{NP}$ -hard [4]).

In a preprocessing phase, two simple data-reduction techniques were applied time and again until no further application is possible:

1. If all trains stopping at station  $S_1$  also stop at station  $S_2$ , then  $S_1$  can be safely removed.

2. If a train  $T_1$  only stops at stations where train  $T_2$  also stops, then  $T_1$  can be safely removed.

The computational study in [14] evaluated this technique on the timetables of several European countries (and their union) and in each case on various selected combinations of train classes. The message of [14] is the impressive result of this study: each of these real-world instances – without any exception! – was reduced to isolated stations and a few, very small non-trivial connected components. Clearly, all isolated stations plus an optimal selection from each non-trivial connected component is an optimal solution to the input instance. A simple brute-force approach is then sufficient.

Due to this extreme result, the general methodological problem is obvious: since this behavior occurred in all real-world instances with negligible variance, “realistic” random instances should also show this behavior. However, if so, they do not give any new insights.

#### Difference 4:

Make statistics on CPU time, operation counts, etc...  
... but what to measure is not always clear.

It is good common practice in statistics to specify the goals of a statistical evaluation before the data is collected. In algorithmics, the favorite candidates are run time (in terms of raw CPU time, operation counts, cache misses, etc.) and space consumption. This is also the usual base for comparisons of algorithms.

However, sometimes the “right” measure to capture the quality of an algorithm is only known afterwards, and it may be specific for a particular algorithmic approach. The second concrete example in the discussion of Difference 3, covering trains by stations, may also serve as an illustration of Difference 4.

In this example, the space consumption and run time of the preprocessing phase and the brute-force kernel are negligible compared to the space requirement of the raw data and the time required for reading the raw data from the background device. What really counts is the number of non-trivial connected components and the distribution of their sizes. Of course, this insight was not anticipated, so the only relevant statistical measures could not be chosen a priori. It goes without saying that these measures are specific for the application and for the chosen algorithmic approach. It is not clear what a reasonable, fair measure for comparisons with other algorithmic approaches could look like.

#### Difference 5:

Make statistics on CPU time, operation counts, etc...  
... but non-quantifiable characteristics are often more important.

Here are three examples of non-quantifiable characteristics:

– *Flexibility:*

The typical algorithmic result in publications from our community is focused on one specific problem, very often a very restricted special case of a general problem (*e.g.* a classical problem such as disjoint-paths or Steiner-tree reduced to a restricted class of graphs). However, in an application project, one cannot assume that the formal model can be once designed in the first phase and is then stable for the rest of the development phase (not to mention the maintenance phase). In Difference 1, two frequently occurring causes were stated:

- The problem in which the “customer” is interested may change (volatile details).
- Our understanding of the (underspecified) problem changes.

Simple, artificial example for illustration: suppose we are faced with a problem on embedded planar graphs, and after a lot of hard theoretical work, we found an equivalent problem in the dual graph, which can be solved incredibly fast by a smart implementation of an ingenious algorithm. Then you feed the implementation with the data provided by the company, and it crashes...

Error tracing reveals that there are edge crossings in the data. You call the responsible manager to complain about the dirtiness of the data. However, the manager will maybe tell you that the graph may indeed be slightly non-planar. Our assumption that the graph always be planar was the result of one of these unavoidable communication problems.

Our imaginary algorithm relied on the dual graph, so it can probably not be generalized to the new situation. In other words, our algorithm was not flexible enough, and chances are high that we have to start our research from scratch again.

– *Error diagnostics:*

Most algorithms in the literature simply return “sorry” in case the input instance is unsolvable. Some algorithms also allow the construction of a certificate for infeasibility. However, the answer “sorry” and a certificate for infeasibility are not very helpful for the end-user. The end-user needs to know what (s)he can do in order to overcome the problem. This information may be quite different from a certificate.

The main example for Difference 1, reconstructing the neighborhood relations among the patches of a CAD model, may also serve as an example for this point. As mentioned above, there is no perspective for a realistic formal model. Nonetheless, algorithmics can make a significant contribution here, if the problem definition is changed slightly in view of error diagnostics.

Since no satisfactory algorithmic solution is available, the result of an algorithm must anyway be corrected by the end-user. Hence, a more realistic objective would be to provide an approximation of the solution which only requires a minor revision effort from the end-user. Of course, a small number of errors is helpful for that, so one could be tempted to simply relax the

problem to its optimization version: errors are admitted but should be kept to a minimum.

However, the revision effort of the end-user is mainly determined by the problem to *find* the errors. This insight gives rise to another objective: construct an (erroneous) solution that can be presented visually such that the errors are easily found by human beings. It has turned out [16] that this problem is indeed treatable. Roughly speaking, it suffices to construct an overestimation of the correct solution. A simple coloring of a picture of the workpiece then allows the end-user to find all errors through a brief glance over the picture. Of course, the number of errors in the overestimation should still be as small as possible. However, this is not the main point anymore. It is more important to ensure that all details are on the “safe” (overestimating) side. To guarantee this,<sup>4</sup> a larger number of errors must potentially be accepted.

To summarize, only the deviation from the classical algorithmic viewpoint (which means that a problem should be solved fully automatically) allowed an algorithmic treatment of the problem.

– *Interactivity:*

An algorithm is not necessarily implemented as a stand-alone module, but is often intended as a core component of a larger software package. Nowadays, software packages are typically interactive, and this may have an impact on the feasibility of an algorithmic approach.

An example is production planning in a factory.<sup>5</sup> Here are three concrete examples of requirements imposed by interactivity.

- *Additional input:*

If a new customer order is accepted, the schedule must be computed again. It is often highly desirable that the schedule does not change significantly. For example, most random approaches might be excluded by this requirement.

- *Additional restrictions imposed interactively:*

For example, for reasons that are outside the underlying formal problem specification, the end-user (the supervising engineer-on-duty) may wish to fix the execution time of a job in advance. For example, a so-called *campaign* is a certain period in which a certain machine is only allowed to execute specific operations. Campaigns are possibly outside the formal model, simply because they are hard to handle algorithmically. To handle them manually, the end-user must be able to fix the execution times of selected operations to the campaign time.

Note that there is a significant difference to the first item: a new customer order is nothing but additional input; time restrictions may not be part

---

<sup>4</sup> Of course, a rigorous mathematical guarantee is beyond our reach, so an empirical “guarantee” must suffice.

<sup>5</sup> This example was taken from on-going work. No quotable manuscript is available so far.



of the original problem definition and thus requires an extension of the model.

- *Fast termination required:*

Sometimes the end-user needs a fairly good solution (which need not be totally feasible) very quickly. For example, the end-user must decide quickly whether an additional customer order can be accepted or not (imagine the customer calls the engineer-on-duty by phone). Then the engineer needs a raw estimation of the production plan. This requires something like an iterative algorithm that produces fairly good intermediate results right from the beginning. In other words, algorithmic approaches that do not have such a property are excluded.

### Difference 6:

Compare the algorithms...

... but the algorithms may solve different specifications of the underspecified problem.

*Mesh refinement* [11,12] is an example of Difference 6. In the CAD design process, this is the next step after the reconstruction of the neighborhood relations (which was the main example for Difference 1 and for item “failure handling and error diagnostics” of Difference 5). The problem is to decompose each patch into quadrilateral patches such that a mathematical analysis (finite-element analysis) is efficient and accurate. This problem is also an example for Difference 1 because it is only roughly understood how algorithmically treatable criteria such as the shapes of the quadrilaterals affect the performance of the finite-element method.

Not surprising, the individual algorithms proposed in the literature are based on various, completely different formal models (which are often not even stated explicitly), and they are tuned in view of the objectives of the corresponding model. A comparison of two or more algorithms must also be based on some formal model. However, there seems to be no “neutral” formal model, which is fair to all approaches.

This problem also occurs silently in comparisons with manual work. Many algorithmic problems from the industry were first solved by hand, and the first algorithmic results are compared to the manual solutions in use. However, many decisions in the manual design of solutions might be due to criteria that have never been made explicit, but were applied “intuitively.” If the comparison of the algorithmic and the manual results is based on the side constraints obeyed by the algorithm and the objective function optimized by the algorithm, the evaluation is inevitably unfair to the manual solution.

## Conclusion

*Meta-heuristics* such as *simulated annealing*, *genetic algorithms*, and *neural networks* are very popular and in wide-spread use outside academia. From personal

communications I conclude that many people in our community look down onto these methods because they do not achieve the performance (not to mention the mathematical elegance) of truly algorithmic solutions.

However, from an engineering viewpoint, things look a bit different. Common meta-heuristics are easy to design, easy to implement, easy to understand (even by non-algorithmicians), and easy to adapt to changes of underspecified or volatile details.<sup>6</sup> Moreover, the ability to solve underspecified problems is inherent in all implicit problem solvers such as neural networks, which learn a problem from examples. Other meta-heuristics, which depend on an exact problem specification, are still flexible enough to allow a very general formal problem framework. In such a framework, variations of the problem are widely expressible as numerical parameters (“little screws”), which can be fixed according to experiments and to heuristical insights that could not be formalized.

In view of applications, it might be promising to look for approaches that combine the best from classical algorithmics – the efficiency – and from the world of meta-heuristics. For example, this could mean the design of efficient algorithms for problem definitions that are broad and flexible enough to cover unanticipated changes. For instance, the result from [12] demonstrates that this is not impossible: it has turned out that a certain reduction to the bidirected-flow approach [1] covers a broad variety of problem variants, which are simply expressed as different settings of the numerical parameters (capacities and cost values).

This is but one example, which was taken from our work just for convenience. There are also many examples from the work of other researchers. However, all of this work is punctual and isolated so far. In my opinion, more systematic research is feasible and promising, and would be a fascinating task.

## References

1. B. Gerards: *Matching*. In M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. and Nemhauser (eds.): *Handbook in Operations Research and Management Science* 7, (*Network Models*), Elsevier, 1995.
2. J. Hooker: *Needed: An Empirical Science of Algorithms*. *Operations Research* (1994), 201–212.
3. J. Hooker: *Testing Heuristics: We Have It All Wrong*. *Journal of Heuristics* (1995), 33–42.
4. M.A. Garey and D.S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
5. D.S. Johnson: *A Theoretician’s Guide to the Experimental Analysis of Algorithms*. <http://www.research.att.com/~dsj/papers/exper.ps>, 1996.
6. H.F. Jackson, P.T. Boggs, S.G. Nash und S. Powell: *Guidelines for Reporting Results of Computational Experiments – Report of the Ad Hoc Committee*. *Mathematical Programming* (1991), 413–425.

<sup>6</sup> Note that Difference 1 – underspecification and volatile details – is closely related to most of the Differences 2–6 and thus fundamental.

7. C.Y. Lee, J. Bard, M.L. Pinedo und W. Wilhelm: *Guidelines for Reporting Computational Results in IIE Transactions*. IEE Transactions (1993), 121–123.
8. C. McGeoch: *Analyzing Algorithms by Simulation: Variance Reduction Techniques and Simulation Speedups*. ACM Computing Surveys (1992), 195–212.
9. C. McGeoch: *Toward an Experimental Method of Algorithm Simulation*. INFORMS Journal on Computing (1996), 1–15.
10. B. Moret: *Experimental Algorithmics: Old Problems and New Directions*. Slides of an invited lecture in: On-Line Proceedings of the 2nd Workshop on Algorithms Engineering (WAE '98), <http://www.mpi-sb.mpg.de/~wae98/PROCEEDINGS/>.
11. R.H. Möhring, M. Müller–Hannemann and K. Weihe: *Mesh Refinement via Bidirected Flows – Modeling, Complexity, and Computational Results*. Journal of the ACM 44 (1997), 395–426.
12. M. Müller–Hannemann and K. Weihe: *On the Discrete Core of Quadrilateral Mesh Refinement*. International Journal on Numerical Methods in Engineering (1999), 593–622.
13. S. Skiena: *The Algorithm Design Manual*. Springer, 1998.
14. K. Weihe: *Covering Trains by Stations or the Power of Data Reduction*. On-Line Proceedings of the 1st Workshop on Algorithms and Experiments (ALEX '98), <http://rtm.science.unitn.it/alex98/proceedings.html>.
15. K. Weihe, U. Brandes, A. Liebers, M. Müller–Hannemann, D. Wagner, and T. Willhalm: *Empirical Design of Geometric Algorithms*. Proceedings of the 15th ACM Symposium on Computational Geometry (SCG '99), 86–94.
16. K. Weihe and T. Willhalm: *Why CAD Data Repair Requires Discrete Techniques*. On-Line Proceedings of the 2nd Workshop on Algorithms Engineering (WAE '98), <http://www.mpi-sb.mpg.de/~wae98/PROCEEDINGS/>.
17. K. Weihe and T. Willhalm: *Reconstructing the Topology of a CAD Model – A Discrete Approach*. Algorithmica 26 (2000), 126–147.

# An Experimental Study of Online Scheduling Algorithms<sup>\*</sup>

Susanne Albers<sup>1</sup> and Bianca Schröder<sup>2</sup>

<sup>1</sup> Lehrstuhl Informatik 2, Universität Dortmund, 44221 Dortmund, Germany  
[albers@cs.uni-dortmund.de](mailto:albers@cs.uni-dortmund.de)

<sup>2</sup> Computer Science Department, Carnegie Mellon University,  
Pittsburgh, PA 15213, USA  
[bianca@cs.cmu.edu](mailto:bianca@cs.cmu.edu)

**Abstract.** We present the first comprehensive experimental study of on-line algorithms for Graham's scheduling problem. In Graham's scheduling problem, which is a fundamental and extensively studied problem in scheduling theory, a sequence of jobs has to be scheduled on  $m$  identical parallel machines so as to minimize the makespan. Graham gave an elegant algorithm that is  $(2 - 1/m)$ -competitive. Recently a number of new online algorithms were developed that achieve competitive ratios around 1.9. Since competitive analysis can only capture the worst case behavior of an algorithm a question often asked is: Are these new algorithms geared only towards a pathological case or do they perform better in practice, too?

We address this question by analyzing the algorithms on various job sequences. We have implemented a general testing environment that allows a user to generate jobs, execute the algorithms on arbitrary job sequences and obtain a graphical representation of the results. In our actual tests, we analyzed the algorithms (1) on real world jobs and (2) on jobs generated by probability distributions. It turns out that the performance of the algorithms depends heavily on the characteristics of the respective work load. On job sequences that are generated by standard probability distributions, Graham's strategy is clearly the best. However, on the real world jobs the new algorithms often outperform Graham's strategy. Our experimental study confirms theoretical results and gives some new insights into the problem. In particular, it shows that the techniques used by the new online algorithms are also interesting from a practical point of view.

## 1 Introduction

During the last ten years *online scheduling* has received a lot of research interest, see for instance [1,2,15,19,20]. In online scheduling, a *sequence of jobs*

---

<sup>\*</sup> Due to space limitations, this extended abstract contains only parts of our results.

A full version of the paper can be obtained at

<http://ls2-www.informatik.uni-dortmund.de/~albers/> or

<http://www.cs.cmu.edu/~bianca/>

$\sigma = J_1, J_2, \dots, J_n$  has to be scheduled on a number of machines. The jobs arrive one by one; whenever a new job arrives, it has to be dispatched immediately to one of the machines, without knowledge of any future jobs. The goal is to optimize a given objective function. Many online algorithms for various scheduling problems have been proposed and evaluated using competitive analysis. However, an experimental evaluation of the algorithms was usually not presented. We remark here that there exist experimental studies for many scheduling strategies used in parallel supercomputers [9,10]. However, these are strategies for scheduling jobs that can span *more than one machine*, while in Graham's model each job has to be assigned to exactly *one* machine. Moreover, Savelsbergh *et al.* [16] recently gave an experimental analysis of *offline* approximation algorithms for the problem of minimizing the weighted sum of completion times.

In this paper we present an experimental study of algorithms for a fundamental problem in online scheduling. This problem is referred to as Graham's problem and has been investigated extensively from a theoretical point of view, see for instance [1,2,3,5,6,7,4,8,12,11,15,18]. In Graham's problem, a sequence of jobs  $\sigma = J_1, J_2, \dots, J_n$  has to be scheduled on  $m$  identical parallel machines. Whenever a new job  $J_t$ ,  $1 \leq t \leq n$ , arrives, its processing time  $p_t$  is known in advance. Each job has to be assigned immediately on one of the machines, without knowledge of any future jobs. The goal is to minimize the *makespan*, which is the completion time of the job that finishes last. This problem arises frequently in high performance and supercomputing environments. Here, it is often the case that either preemption is not supported by the system or the high memory requirements of the jobs make preemption prohibitively expensive. The runtimes of the jobs are known at least approximately since users are usually required to give an estimate for the CPU requirements of their jobs. The objective of minimizing the makespan translates in this setting to achieving a high utilization on the machines. In addition to its practical relevance, Graham's problem is important because it is the root of many problem variants where, for instance, preemption is allowed, precedence constraints exist among jobs, or machines run at different speeds.

In 1966 Graham gave an algorithm that is  $(2 - 1/m)$ -competitive. Following [17] we call an online scheduling algorithm *c-competitive* if, for all job sequences  $\sigma = J_1, J_2, \dots, J_n$ ,  $A(\sigma) \leq c \cdot OPT(\sigma)$ , where  $A(\sigma)$  is the makespan of the schedule produced by  $A$  and  $OPT(\sigma)$  is the makespan of an optimal schedule for  $\sigma$ . It was open for a long time whether an online algorithm can achieve a competitive ratio that is asymptotically smaller than 2, for all values of  $m$ . In the early nineties Bartal *et al.* [2] presented an algorithm that is 1.986-competitive. Karger *et al.* [15] generalized the algorithm and gave an upper bound of 1.945. Recently, Albers [1] presented an improved algorithm that is 1.923-competitive. An interesting question is whether these new techniques are geared only towards a pathological worst case or whether they also lead to better results in practice. In this paper we address this question and present the first comprehensive experimental study of online algorithms for Graham's scheduling problem.

**The Testing Environment:** We have implemented a general testing environment that allows a user to generate jobs, execute the algorithms on user-defined or randomly generated job files and obtain a graphical representation of the results. The environment can be easily modified to test algorithms for other online scheduling problems as well.

**Description of the Experiments:** We implemented the online algorithms by Graham, Bartal *et al.*, Karger *et al.* and Albers and tested them on (1) real world job sequences as well as on (2) job sequences generated by probability distributions. As for the real world jobs, we investigated data sets from three different machine configurations. The first data set consists of job sequences taken from the log files of three *MPP's* (Massively Parallel Processors) at three different supercomputing centers. The runtimes in the second data set were extracted from a log file of a 16 processor *vector machine* at the Pittsburgh Supercomputing Center. This environment resembles very much the model described above. The jobs in the third data set were obtained from a process accounting on a Sun Ultra *workstation*. This workstation is one of the main computing servers at the Max Planck Institute in Saarbrücken. We believe that an analysis of the algorithms' performance on real job traces gives the most meaningful results. However, we also evaluated the algorithms under job sequences generated by probability distributions. More specifically, we generated job sequences according to the uniform, exponential, Erlang, hyperexponential and Bounded Pareto distributions.

For each job sequence and each of the four algorithms, we determined the ratio *online makespan/optimum makespan* after each scheduling step, i.e. whenever a new job was scheduled, the ratio was re-computed. This allows us not only to compare the algorithms against each other but also gives us a measure for how far away the online algorithms are from the optimal offline solution at any given point of time. Finally, we also considered the algorithms' performance for different machine numbers and evaluated settings with 10, 50, 100 and 500 machines.

**Summary of the Experimental Results:** The results differ substantially depending on the workload characteristics. In the experiments with real world jobs, the ratios *online makespan/optimum makespan* fluctuate. We observe sudden increases and decreases, depending on the size of the last job that was scheduled. Whenever the processing time of a new job is in the order of the average load on the machines, the ratio goes up, with values up to 1.8–1.9. Whenever the processing time of a new job is very large compared to the average load on the machines, the ratio drops and approaches 1. Only after a large number of jobs have been scheduled do the ratios stabilize. An important result of the experiments is that some of the new algorithms suffer much less from these sudden increases than Graham's algorithm and therefore lead to a more predictable performance. They also often outperform Graham's algorithm. This makes the new algorithms also interesting from a practical point of view.

In the experiments with job sequences generated by one of the standard probability distributions, the ratios *online makespan/optimum makespan* con-

verge quickly. Graham's algorithm outperforms the other three algorithms and achieves ratios close to 1. The ratios of the algorithm by Bartal *et al.* and Albers are slightly higher and converge to values between 1.2 and 1.3. The algorithm by Karger *et al.* performs worse, with ratios between 1.7 and 1.9. Surprisingly, these results show for all standard probability distributions.

Our experimental study confirms and validates theoretical results. It shows that the new online algorithms for Graham's problem are also interesting from a practical point of view.

**Organization of the Paper:** In Section 2 we describe the online scheduling algorithms by Graham, Bartal *et al.*, Karger *et al.* and Albers. The testing environment is briefly described in Section 3. In Section 4 we give a detailed presentation of the experiments with real world jobs. A description of the tests with randomly generated jobs follows in Section 5.

## 2 The Algorithms

In this section we describe the online algorithms that we will analyze experimentally. An algorithm is presented with a job sequence  $\sigma = J_1, J_2, \dots, J_n$ . Let  $p_t$  denote the processing time of  $J_t$ ,  $1 \leq t \leq n$ . At any time let the *load* of a machine be the sum of the processing times of the jobs already assigned to it. In the following, when describing the algorithms, we assume that an algorithm has already scheduled the first  $t-1$  jobs  $J_1, \dots, J_{t-1}$ . We specify how the next job  $J_t$  is scheduled.

**Algorithm by Graham:** Schedule  $J_t$  on the machine with the smallest load.

All the other algorithms maintain a list of the machines sorted in non-decreasing order by current load. The goal is to always maintain some lightly loaded and some heavily loaded machines. Let  $M_i^{t-1}$  denote the machine with the  $i$ -th smallest load,  $1 \leq i \leq m$ , after exactly  $t-1$  jobs have been scheduled. In particular,  $M_1^{t-1}$  is the machine with the smallest load and  $M_m^{t-1}$  is the machine with the largest load. We denote by  $l_i^{t-1}$  the load of machine  $M_i^{t-1}$ ,  $1 \leq i \leq m$ . Note that the load  $l_m^{t-1}$  of the most loaded machine is always equal to the current makespan. Let  $A_i^{t-1}$  be the average load on the  $i$  smallest machines after  $t-1$  have been scheduled. The algorithm by Bartal *et al.* keeps about 44.5% of the machines lightly loaded.

**Algorithm by Bartal *et al.*:** Set  $k = \lceil 0.445m \rceil$  and  $\epsilon = 1/70$ . Schedule  $J_t$  on  $M_{k+1}^{t-1}$  if  $l_{k+1}^{t-1} + p_t \leq (2 - \epsilon)A_k^{t-1}$ . Otherwise schedule  $J_t$  on the machine with the smallest load.

The algorithm by Karger *et al.* maintains a full stair-pattern.

**Algorithm by Karger *et al.*:** Set  $\alpha = 1.945$ . Schedule  $J_t$  on the machine  $M_k^{t-1}$  with the largest load such that  $l_k^{t-1} + p_t \leq \alpha A_{k-1}^{t-1}$ . If there is no such machine, then schedule  $J_t$  on the machine with the smallest load.

The algorithm by Albers keeps 50% of the machines lightly loaded.

**Algorithm by Albers:** Set  $c = 1.923$ ,  $k = \lfloor \frac{m}{2} \rfloor$  and  $j = 0.29m$ . Set  $\alpha = \frac{(c-1)k-j/2}{(c-1)(m-k)}$ . Let  $L_l$  be the sum of the loads on machines  $M_1^t, \dots, M_k^t$  if  $J_t$  is scheduled on the least loaded machine. Similarly, let  $L_h$  be the sum of the loads on machines  $M_{k+1}^t, \dots, M_m^t$  if  $J_t$  is scheduled on the least loaded machine. Let  $\lambda_m^t$  be the makespan, i.e. the load of the most loaded machine, if  $J_t$  is scheduled on the machine with the  $(k+1)$ -st smallest load. Recall that  $l_m^{t-1}$  is the makespan before the assignment of  $J_t$ . Schedule  $J_t$  on the least loaded machine if one of the following conditions holds: (a)  $L_l \leq \alpha L_h$ ; (b)  $\lambda_m^t > l_m^{t-1}$  and  $\lambda_m^t > c \cdot \frac{L_l + L_h}{m}$ . Otherwise schedule  $J_t$  on the machine with the  $(k+1)$ -st smallest load.

### 3 The Testing Environment

This section describes briefly the functionality of the program that we have implemented to compare the performance of the algorithms. After being started, the program opens a window consisting of a panel section for user interaction and a drawing section for the graphical representation of the scheduling algorithms. The main functionality of the program is provided by three buttons labeled *File*, *Execute* and *Generate Jobs* in the panel section. The *File* button provides general functions such as saving the graphical representation of an algorithm's execution as a Postscript file. The *Generate Jobs* button allows the user to generate jobs from the uniform, exponential, hyperexponential and Erlang distribution according to parameters defined by the user or to enter jobs manually. The *Execute* button provides the interface to the scheduling algorithms. After specifying the algorithm, the number of machines and the job file, the user can choose whether he wants to see only the result after all jobs are scheduling or whether he wants to watch the entire scheduling process.

### 4 Experiments with Real World Jobs

Before we discuss the results of the experiments we describe the experimental setup. The jobs used in the experiments come from three different types of systems. The first data set consists of job traces taken from *MPP's* (massively parallel processors) and were obtained from Feitelson's Parallel Workloads Archive. It includes a trace from a 512-node IBM-SP2 at Cornell Theory Center (CTC), a trace from a 100-node IBM-SP2 at the KTH in Sweden and a trace from a 128-node iPSC/860 at NASA Ames. The second data set consists of runtimes measured at the Pittsburgh Supercomputing Center's Cray C90, which is a *vector machine*. The jobs in the third data set were obtained from a process accounting on a Sun Ultra *workstation* with two 200 MHz processors and 1024 MB main memory. This workstation is one of the main computing servers at the Max Planck Institute in Saarbrücken. The following table summarizes the main characteristics of the workloads. These will be crucial for the interpretation of the results.



System	Year	Number of Jobs	Mean Size (sec)	Min (sec)	Max (sec)	Squared Coefficient of Variation
CTC IBM-SP2	1996 - 1997	57290	2903.6	1	43138	2.72
KTH IBM-SP2	1996 - 1997	28490	8878.9	1	226709	5.48
NASA Ames iPSC/860	1993	42050	348.20	1	62643	27.21
PSC Cray C90	1997	54962	4562.6	1	2222749	43.16
MPI Sun Ultra	1998	300000	2.3	0.01	47565.4	7550.58

We split each job trace into job sequences containing 10000 jobs. We then ran the online algorithms on each job sequence and recorded the ratio *online makespan/optimum makespan* after each job. The machine numbers used in these experiments range from 10 to 500. The next two sections describe and analyze the experimental results. Due to space limitations, we can only present the results for 10 machines. The results for larger machine numbers are given in the full paper.

#### 4.1 The Experimental Results

We begin with the results for the MPP data. The development of the ratios under the job sequences obtained from the CTC and the KTH traces was virtually identical. Figure 1 shows the typical development of the ratios of the online algorithms' makespans to the optimal makespans for these job sequences. We

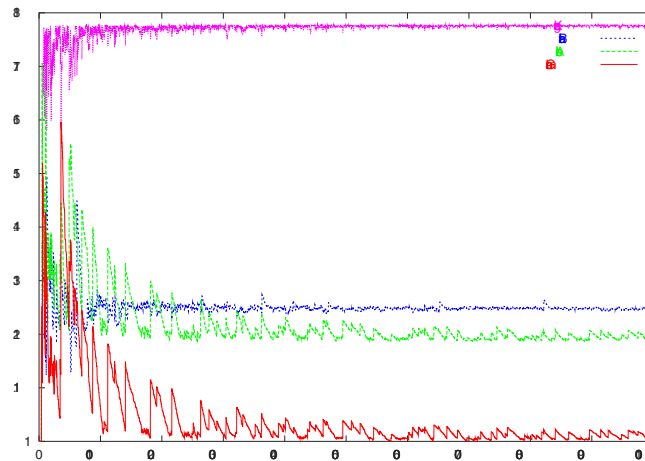
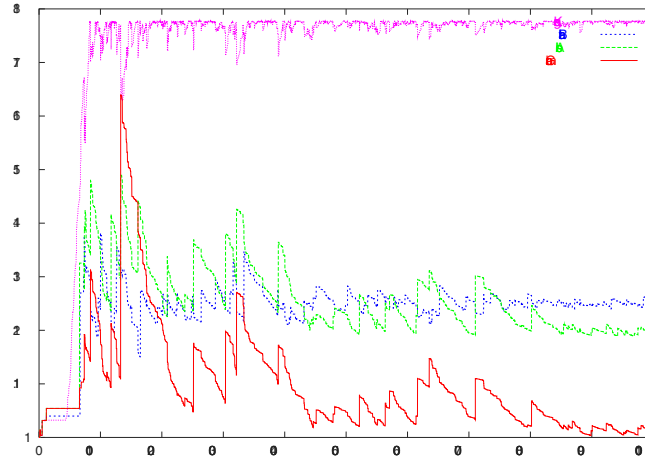


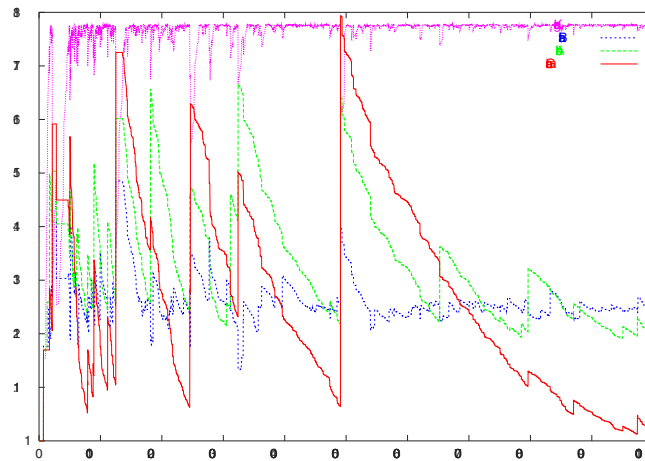
Fig. 1. Performance of the online algorithms on the KTH data

see that the ratios during the first 1000 jobs oscillate between values of 1.1 and 1.7. The only exception are the ratios for Karger's algorithms which immediately approach a value of 1.8. After the first 1000 jobs the ratios of all algorithms stabilize. For Bartal's and Albers' algorithm they converge towards a value around 1.2 while the ratio for Graham's algorithm approaches 1. Figure 2 shows the results for the NASA jobs. The general tendencies in the development of the ratios



**Fig. 2.** Performance of the online algorithm on the NASA data

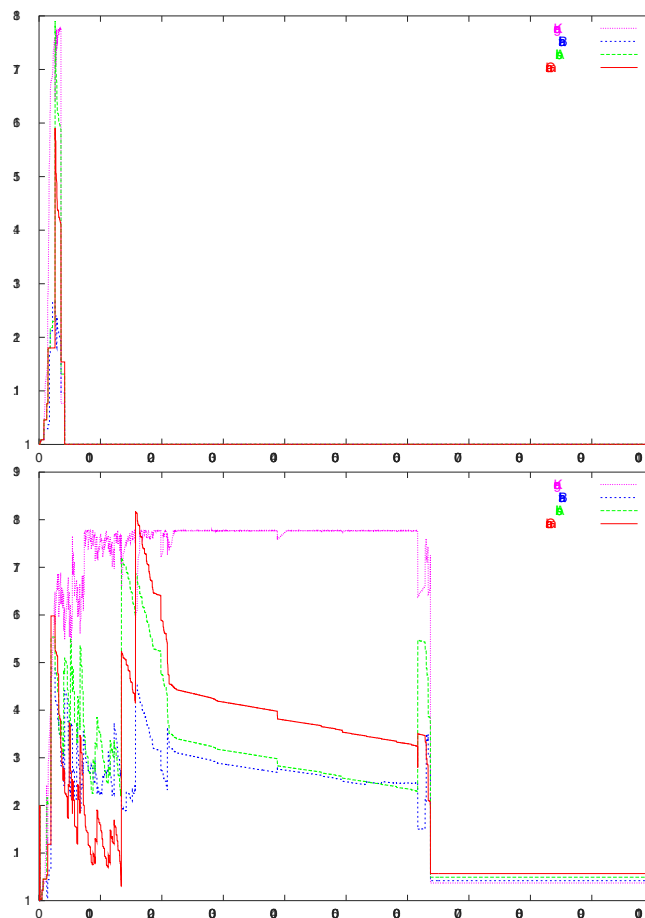
is similar to that observed for the CTC and the KTH data. Initially, the ratios fluctuate until they finally converge to the same values as for the CTC/KTH data. In contrast to the results for the CTC and KTH jobs it takes much longer until the ratios stabilize. Under the PSC data the ratios are even more volatile (see Figure 3). Especially, the ratio for Graham's algorithm is extremely instable



**Fig. 3.** Performance of the online algorithms on the PSC data

and goes frequently up to values between 1.7 and 1.8. Bartal's algorithm, on the other hand, converges very early to a ratio close to 1.2. After around 9000 jobs have been scheduled the ratios approach the values that we observed for the previous traces. The workstation data set is the only one where the results were different for the various job sequences. They also differ from the results we have

observed so far. Figure 4 shows two typical scenarios for job sequences extracted from the workstation trace. We see that the ratios again oscillate in the beginning, but this time they don't converge gradually to some value. Instead they



**Fig. 4.** Typical results for experiments with workstation jobs

drop very abruptly to 1 and don't change after that. This sudden drop in the ratios can occur very early as shown in Figure 4 (top) or later in the scheduling process as in Figure 4 (bottom).

## 4.2 Analysis of the Results

To interpret the experimental results it is helpful to understand in which way a new job can affect the ratio *online makespan/optimal makespan*. Depending on its size compared to the average load on the machines, a job can have one of the following three effects.

1. If the size of an arriving job is small compared to the average load on the machines, the job will neither significantly affect the optimum makespan nor the online makespans. Therefore, the ratio *online makespan/optimal makespan* will remain almost unchanged.
2. If the size of an arriving job is in the order of the average load on the machines the ratio *online makespan/optimal makespan* will increase. The reason is that all algorithms have to maintain a certain balance between the load on the machines to prevent the makespan from growing too large. Therefore, they will have to assign the arriving job to a machine that contains already an amount of load close to the average load. The optimal offline strategy would have been to reserve one machine almost entirely for this job. Therefore, if the size of a new job is approximately that of the average load on the machines, the ratio *online makespan/optimal makespan* will increase and in the worst case approach 2.
3. If the size of the new job is extremely large compared to the average load on the machines, the new job will completely dominate the optimal makespan, as well as the makespan of an online algorithm. This leads to almost the same makespan for the optimal and the online algorithm's solutions. As a result, the ratio *online makespan/optimal makespan* will approach 1.

In the following we will refer to these three effects as effect 1, 2, and 3, respectively. Note at this point that a sequence of small jobs (effect 1) followed by a job triggering effect 2 is the worst case scenario for Graham's algorithm. This is because Graham will distribute the small jobs completely evenly over the machines and therefore has to assign the "effect 2 job" to a machine that contains already a lot of load. All the other algorithms try to alleviate this problem by keeping some of the machines lightly loaded and hence reserving some space for "effect 2 jobs" that might arrive in the future.

How likely the occurrence of each of the three effects is and how pronounced the effect will be, depends on the characteristics of the workload and the scheduling algorithm. If the variability in the job sizes is low effect 2 and 3 are very unlikely to occur. The reason is that a low variability in the job size distribution means that the jobs are relatively similar in size. Therefore, the probability that a new job has a size similar to that of all the jobs at one machine combined is very low. Looking at the table with the characteristics of the traces we see that the CTC and the KTH traces have a very low squared coefficient of variation, which indicates a low variability in the job sizes. This explains why the ratios converged so quickly in the experiments with these traces: the low variability in the job sizes makes the arrival of an "effect 2" or "effect 3" job very unlikely. It also explains why the performance of the three new algorithms is worse than that of Graham's algorithm (except for the first jobs). The new algorithms reserve some space for large jobs that never arrive and therefore have higher makespans. For the NASA and the PSC trace the squared coefficient of variation is much higher than for the CTC and the KTH traces indicating a higher variability in the job sizes. Therefore, effect 3 and in particular effect 2 are likely to happen, even after many jobs have been scheduled. This leads to the fluctuation of the

*online makespan/optimal makespan* that we observed in Figure 2 and 3. We also see that in this case the strategy of keeping some machines lightly loaded can pay off. The ratios for Bartal’s algorithm, for instance, are in many cases much lower than the ratios of Graham’s algorithm. Moreover, the ratio under Bartal’s algorithms converges quickly leading to a more predictable performance than the heavily oscillating ratio of Graham’s algorithm. In the workstation trace the variability is extremely high meaning that some jobs have a size that is extremely large compared to that of an average job. Typically, in workstation traces the largest 1 percent of all jobs make up half of the total load (a property sometimes referred to as heavy-tailed property). As soon as one of these extremely large jobs arrives, it completely dominates both the optimal and the online makespan. This leads to the drop of the ratios to 1 that we see in Figure 4.

## 5 Experiments with Jobs Generated by Probability Distributions

We also analyzed the performance of the scheduling algorithms on job sequences generated by the following probability distributions: (a) the uniform distribution; (b) the exponential distribution; (c) the Erlang distribution; (d) the hyperexponential distribution; and (e) the Bounded Pareto distribution. When choosing the parameters of the distributions from which the numbers were generated we tried on the one hand to cover a great range and on the other hand to use parameters similar to that in tests presented in [9] and [10]. Details are given in the full version of the paper. The distributions commonly used to model service times of computer systems are the exponential, hyperexponential and the Erlang distribution [14]. For the sake of completeness we also included the uniform distribution. The experimental results for these four standard distributions are discussed in Section 5.1. The Bounded Pareto distribution is discussed in Section 5.2.

### 5.1 The Standard Distributions

Surprisingly, the results did not differ significantly for the various standard distributions. Even more surprisingly, the results were similar for all parameters. Figure 5 shows the development of the ratio *online makespan/optimum makespan* for exponentially distributed job sizes, but also represents the results for the other distributions quite well. We observe that the curves fluctuate to a much smaller degree than under the real work loads. They converge to the same values as in the case of real job sequences, but they do so much faster. The reason is that the variability in the job sizes is much lower for these distributions. The exponential distribution has a squared coefficient of variation of 1 independently of its mean. The Erlang distribution and the uniform distribution always have a squared coefficient of variation less than or equal to 1, independently of how their parameters are chosen. For the hyperexponential distribution it is theoretically possible to choose the parameters as to match the mean and the

squared coefficient of variation of any distribution. However, to achieve squared coefficients of variations as observed for the more variable real world traces one would have set the parameters to very extreme values.

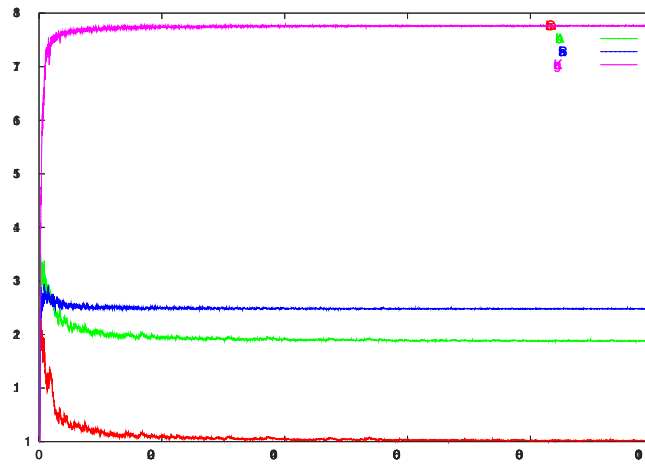


Fig. 5. The performance of the algorithms under an exponential workload

## 5.2 The Bounded Pareto Distribution

In contrast to the standard distributions, the Bounded Pareto distribution can be easily fit to observed data. We chose the parameters for this distribution so as to match the mean job sizes in the various job traces and to create different degrees of variability in the job sizes. It turned out that for a very low variability the results were virtually identical to those for the CTC and the KTH data as shown in Figure 1. For medium variability the results looked very similar to those for the PSC data (see Figure 3). For extremely variable job sizes the results matched those for the workstation traces (see Figure 4). This confirms our theory from Section 4 that the variability in the job sizes is the crucial factor for the performance of the algorithms.

## 6 Conclusion

We saw that the performance of scheduling algorithms depends heavily on the workload characteristics. For workloads with a low variability the simple greedy algorithm by Graham has the best performance. For highly variable real workloads, however, the new algorithms often outperform Graham's algorithm. Our results also show the importance of choosing the right workload when evaluating scheduling algorithms experimentally. In particular, we observed that standard probability distributions do often not capture important characteristics of real workloads very well.

## References

1. S. Albers. Better bounds for online scheduling. In *Proc. 29th Annual ACM Symposium on Theory of Computing*, pages 130–139, 1997.
2. Y. Bartal, A. Fiat, H. Karloff and R. Vohra. New algorithms for an ancient scheduling problem. *Journal of Computer and System Sciences*, 51:359–366, 1995.
3. Y. Bartal, H. Karloff and Y. Rabani. A better lower bound for on-line scheduling. *Information Processing Letters*, 50:113–116, 1994.
4. B. Chen, A. van Vliet and G.J. Woeginger. Lower bounds for randomized online scheduling. *Information Processing Letters*, 51:219–222, 1994.
5. E.G. Coffman, L. Flatto, M.R. Garey and R.R. Weber, Minimizing expected makespans on uniform processor systems, *Adv. Appl. Prob.*, 19:177-201, 1987.
6. E.G. Coffman, L. Flatto and G.S. Lueker Expected makespans for largest-first multiprocessor scheduling, *Performance '84*, Elsevier Science Publishers, 491-506, 1984.
7. E.G. Coffman Jr., G.N. Frederickson and G.S. Lueker. Expected makespans for largest-first sequences of independent tasks on two Processors, *Math. Oper. Res.*, 9:260-266, 1984.
8. U. Faigle, W. Kern and G. Turan. On the performance of on-line algorithms for particular problems. *Acta Cybernetica*, 9:107–119, 1989.
9. D.G. Feitelson and L. Rudolph, editors. *Job Scheduling Strategies for Parallel Processing (IPPS 95). Workshop, Santa Barbara, CA, USA, 25. April, 1995: Proceedings, Springer Lecture Notes in Computer Science*, Volume 949, 1995.
10. D.G. Feitelson and L. Rudolph, editors. *Job Scheduling Strategies for Parallel Processing (IPPS 96). Workshop, Honolulu, Hawaii, April 16, 1996: Proceedings, Springer Lecture Notes in Computer Science*, Volume 1162, 1996.
11. G. Galambos and G. Woeginger. An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling. *SIAM Journal on Computing*, 22:349–355, 1993.
12. R.L. Graham. Bounds for certain multi-processing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
13. L.A. Hall, D.B. Shmoys and J. Wein. Scheduling to minimize average completion time: Off-line an on-line algorithms. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 142–151, 1996.
14. R. Jain. *The Art of Computer Systems Performance Analysis*, Wiley, 1991.
15. D.R. Karger, S.J. Phillips and E. Torng. A better algorithm for an ancient scheduling problem. *Journal of Algorithms*, 20:400–430, 1996.
16. M.W.P. Savelsbergh, R.N. Uma and J. Wein. An experimental study of LP-based approximation algorithms for scheduling problems. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 453–462, 1998.
17. D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
18. J. Sgall. A lower bound for randomized on-line multiprocessor scheduling, *Information Processing Letters*, 63:51–55, 1997.
19. J. Sgall. On-line scheduling. In *Online algorithms: The state of the art*, A. Fiat and G.J. Woeginger. Springer Lecture Notes in Computer Science, Volume 1224, pages 196–231, 1998.
20. D. Shmoys, J. Wein and D.P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24:1313-1331, 1995.

# Implementation of $O(nm \log n)$ Weighted Matchings in General Graphs. The Power of Data Structures

Kurt Mehlhorn and Guido Schäfer

Max-Planck-Institut für Informatik,  
Im Stadtwald, 66123 Saarbrücken, Germany  
{mehlhorn,schaefer}@mpi-sb.mpg.de

**Abstract.** We describe the implementation of an  $O(nm \log n)$  algorithm for weighted matchings in general graphs. The algorithm is a variant of the algorithm of Galil, Micali, and Gabow [12] and requires the use of concatenable priority queues. No previous implementation had a worst-case guarantee of  $O(nm \log n)$ . We compare our implementation to the experimentally fastest implementation (called Blossom IV) due to Cook and Rohe [4]; Blossom IV is an implementation of Edmonds' algorithm and has a running time no better than  $\Omega(n^3)$ . Blossom IV requires only very simple data structures. Our experiments show that our new implementation is competitive to Blossom IV.

## 1 Introduction

The *weighted matching problem* asks for the computation of a *maximum-weight matching* in a graph. A *matching*  $M$  in a graph  $G = (V, E)$  is a set of edges no two of which share an endpoint. The edges of  $G$  have weights associated with them and the weight of a matching is simply the sum of the weights of the edges in the matching. There are two variants of the matching problem: one can either ask for a *maximum-weight perfect matching* (a matching is *perfect* if all vertices in the graph are matched) or for a *maximum-weight matching*. Our implementation can be asked to compute either a maximum-weight perfect matching or a maximum-weight matching. Previous implementations were restricted to compute optimal perfect matchings.<sup>1</sup>

Edmonds [9] invented the famous blossom-shrinking algorithm and showed that weighted perfect matchings can be computed in polynomial time. A straightforward implementation of his algorithm runs in time  $O(n^2m)$ , where  $n$  and  $m$  are the number of vertices and edges of  $G$ , respectively. Lawler [14] and Gabow [10] improved the running time to  $O(n^3)$ . The currently most efficient

---

<sup>1</sup> The maximum-weight matching problem can be reduced to the maximum-weight perfect matching problem: the original graph is doubled and zero weight edges are inserted from each original vertex to the corresponding doubled vertex. For an original graph with  $n$  vertices and  $m$  edges, the reduction doubles  $n$  and increases  $m$  by  $m + n$ ; it soon becomes impractical for large instances.



codes implement variants of these algorithms, Blossom IV of Cook and Rohe [4] being the most efficient implementation available.

Galil, Micali, and Gabow [12] improved the running time to  $O(nm \log n)$  and Gabow [11] achieved  $O(n(m + n \log n))$ . Somewhat better asymptotic running times are known for integral edge weights. The improved asymptotics comes mainly through the use of sophisticated data structures; for example, the algorithm of Galil, Micali, and Gabow requires the use of concatenable priority queues in which the priorities of certain subgroups of vertices can be changed by a single operation. It was open and explicitly asked in [2] and [4], whether the use of sophisticated data structures helps in practice. We answer this question in the affirmative.

The preflow–push method for maximum network flow is another example of an algorithm whose asymptotic running time improves dramatically through the use of sophisticated data structures. With the highest level selection rule and only simple data structures the worst–case running time is  $O(n^2 \sqrt{m})$ . It improves to  $\tilde{O}(nm)$  with the use of dynamic trees. None of the known efficient implementations [3,15] uses sophisticated data structures and attempts to use them produced far inferior implementations [13].

This paper is organized as follows: in Section 2 we briefly review Edmonds’ blossom–shrinking algorithm, in Section 3 we describe our implementation, in Section 4 we report our experimental findings, and in Section 5 we offer a short conclusion.

## 2 Edmonds’ Algorithm

Edmonds’ blossom–shrinking algorithm is a primal–dual method based on a linear programming formulation of the maximum–weight perfect matching problem. The details of the algorithm depend on the underlying formulation. We will first give the formulation we use and then present all details of the resulting blossom–shrinking approach.

*LP Formulations:* Let  $G = (V, E)$  be a general graph. We need to introduce some notions first. An incidence vector  $x$  is associated with the edges of  $G$ :  $x_e$  is set to 1, when  $e$  belongs to the matching  $M$ ; otherwise,  $x_e$  is set to 0. For any subset  $S \subseteq E$ , we define  $x(S) = \sum_{e \in S} x_e$ . The edges of  $G$  having both endpoints in  $S \subseteq V$  are denoted by  $\gamma(S) = \{uv \in E : u \in S \text{ and } v \in S\}$ , and the set of all edges having exactly one endpoint in  $S$  is referred to by  $\delta(S) = \{uv \in E : u \in S \text{ and } v \notin S\}$ . Moreover, let  $\mathcal{O}$  consist of all non–singleton odd cardinality subsets of  $V$ :  $\mathcal{O} = \{\mathcal{B} \subseteq V : |\mathcal{B}| \text{ is odd and } |\mathcal{B}| \geq 3\}$ .

The maximum–weight perfect matching problem for  $G$  with weight function  $w$  can then be formulated as a linear program:

$$\begin{aligned}
 (\text{WPM}) \quad & \text{maximize} && w^T x \\
 & \text{subject to} && x(\delta(u)) = 1 && \text{for all } u \in V, && (1) \\
 & && x(\gamma(\mathcal{B})) \leq \lfloor |\mathcal{B}|/2 \rfloor && \text{for all } \mathcal{B} \in \mathcal{O}, && (2) \\
 & && x_e \geq 0 && \text{for all } e \in E. && (3)
 \end{aligned}$$

(WPM)(1) states that each vertex  $u$  of  $G$  must be matched and (WPM)(2)–(3) assure each component  $x_e$  to be either 0 or 1.

An alternative formulation exists. In the alternative formulation, the second constraint (WPM)(2) is replaced by  $x(\delta(\mathcal{B})) = 1$  for all  $\mathcal{B} \in \mathcal{O}$ . Both the implementation of Applegate and Cook [2] and the implementation of Cook and Rohe [4] use the alternative formulation.

The formulation above is used by Galil, Micali and Gabow [12] and seems to be more suitable to achieve  $O(nm \log n)$  running time. It has the additional advantage that changing the constraint (WPM)(1) to  $x(\delta(u)) \leq 1$  for all  $u \in V$  gives a formulation of the non-perfect maximum-weight matching problem. Our implementation handles both variants of the problem.

Consider the dual linear program of (WPM). A *potential*  $y_u$  and  $z_{\mathcal{B}}$  is assigned to each vertex  $u$  and non-singleton odd cardinality set  $\mathcal{B}$ , respectively.

$$\begin{aligned}
 (\overline{\text{WPM}}) \quad & \text{minimize} \quad \sum_{u \in V} y_u + \sum_{\mathcal{B} \in \mathcal{O}} \lfloor |\mathcal{B}|/2 \rfloor z_{\mathcal{B}} \\
 & \text{subject to} \quad z_{\mathcal{B}} \geq 0 \quad \text{for all } \mathcal{B} \in \mathcal{O}, \\
 & \quad y_u + y_v + \sum_{\substack{\mathcal{B} \in \mathcal{O} \\ uv \in \gamma(\mathcal{B})}} z_{\mathcal{B}} \geq w_{uv} \text{ for all } uv \in E.
 \end{aligned} \tag{1} \tag{2}$$

The *reduced cost*  $\pi_{uv}$  of an edge  $uv$  with respect to a dual solution  $(y, z)$  of  $(\overline{\text{WPM}})$  is defined as given below. We will say an edge  $uv$  is *tight*, when its reduced cost  $\pi_{uv}$  equals 0.

$$\pi_{uv} = y_u + y_v - w_{uv} + \sum_{\substack{\mathcal{B} \in \mathcal{O} \\ uv \in \gamma(\mathcal{B})}} z_{\mathcal{B}}.$$

*Blossom-Shrinking Approach:* Edmonds' blossom-shrinking algorithm is a primal-dual method that keeps a primal (not necessarily feasible) solution  $x$  to (WPM) and also a dual feasible solution  $(y, z)$  to  $(\overline{\text{WPM}})$ ; the primal solution may violate (WPM)(1). The solutions are adjusted successively until they are recognized to be optimal. The optimality of  $x$  and  $(y, z)$  will be assured by the feasibility of  $x$  and  $(y, z)$  and the validity of the complementary slackness conditions (CS)(1)–(2):

$$\begin{aligned}
 (\text{CS}) \quad & x_{uv} > 0 \implies \pi_{uv} = 0 \quad \text{for all edges } uv \in E, \\
 & z_{\mathcal{B}} > 0 \implies x(\gamma(\mathcal{B})) = \lfloor |\mathcal{B}|/2 \rfloor \text{ for all } \mathcal{B} \in \mathcal{O}.
 \end{aligned} \tag{1} \tag{2}$$

(CS)(1) states that matching edges must be tight and (CS)(2) states that non-singleton sets  $\mathcal{B}$  with positive potential are *full*, i.e., a maximum number of edges in  $\mathcal{B}$  are matched.

The algorithm starts with an arbitrary matching  $M$ .<sup>2</sup>  $x$  satisfies (WPM)(2)–(3). Each vertex  $u$  has a potential  $y_u$  associated with it and all  $z_{\mathcal{B}}$ 's are 0. The

<sup>2</sup> We will often use the concept of a matching  $M$  and its incidence vector  $x$  interchangeably.

potentials are chosen such that  $(y, z)$  is dual feasible to  $(\overline{\text{WPM}})$  and, moreover, satisfies (CS)(1)–(2) with respect to  $x$ . The algorithm operates in phases.

In each phase, two additional vertices get matched and hence do no longer violate (WPM)(1). After  $O(n)$  phases, either all vertices will satisfy (WPM)(1) and thus the computed matching is optimal, or it has been discovered that no perfect matching exists.

The algorithm attempts to match free vertices by growing so-called *alternating trees* rooted at free vertices.<sup>3</sup> Each alternating tree  $T_r$  is rooted at a free vertex  $r$ . The edges in  $T_r$  are tight and alternately matched and unmatched with respect to the current matching  $M$ . We further assume a labeling for the vertices of  $T_r$ : a vertex  $u \in T_r$  is labeled *even* or *odd*, when the path from  $u$  to  $r$  is of even or odd length, respectively. Vertices that are not part of any alternating tree are matched and said to be *unlabeled*. We will use  $u^+$ ,  $u^-$  or  $u^\varnothing$  to denote an even, odd or unlabeled vertex.

An alternating tree is extended, or *grown*, from even labeled tree vertices  $u^+ \in T_r$ : when a tight edge  $uv$  from  $u$  to a non-tree vertex  $v^\varnothing$  exists, the edge  $uv$  and also the matching edge  $vw$  of  $v$  (which must exist, since  $v$  is unlabeled) is added to  $T_r$ . Here,  $v$  and  $w$  get labeled odd and even, respectively.

When a tight edge  $uv$  with  $u^+ \in T_r$  and  $v^+ \in T_{r'}$  exists, with  $T_r \neq T_{r'}$ , an *augmenting path* from  $r$  to  $r'$  has been discovered. A path  $p$  from a free vertex  $r$  to another free vertex  $r'$  is called *augmenting*, when the edges along  $p$  are alternately in  $M$  and not in  $M$  (the first and last edge are unmatched). Let  $p_r$  denote the tree path in  $T_r$  from  $u$  to  $r$  and, correspondingly,  $p_{r'}$  the tree path in  $T_{r'}$  from  $v$  to  $r'$ . By  $\overline{p_r}$  we denote the path  $p_r$  in reversed order. The current matching  $M$  is augmented by  $(\overline{p_r}, uv, p_{r'})$ , i.e., all non-matching edges along that path become matching edges and vice versa. After that, all vertices in  $T_r$  and  $T_{r'}$  will be matched; therefore, we can destroy  $T_r$  and  $T_{r'}$  and unlabeled all their vertices.

Assume a tight edge  $uv$  connecting two even tree vertices  $u^+ \in T_r$  and  $v^+ \in T_r$  exists (in the same tree  $T_r$ ). We follow the tree paths from  $u$  and  $v$  towards the root until the lowest common ancestor vertex  $lca$  has been found.  $lca$  must be even by construction of  $T_r$  and the simple cycle  $C = (lca, \dots, u, v, \dots, lca)$  is full. The subset  $\mathcal{B} \subseteq V$  of vertices on  $C$  are said to form a *blossom*, as introduced by Edmonds [9]. A key observation is that one can *shrink* blossoms into an even labeled pseudo-vertex, say  $lca$ , and continue the growth of the alternating trees in the resulting graph.<sup>4</sup> To shrink a cycle  $C$  means to collapse all vertices of  $\mathcal{B}$  into a single pseudo-vertex  $lca$ . All edges  $uv$  between vertices of  $\mathcal{B}$ , i.e.,  $uv \in \gamma(\mathcal{B})$ , become non-existent and all edges  $uv$  having exactly one endpoint  $v$  in  $\mathcal{B}$ , i.e.,  $uv \in \delta(\mathcal{B})$ , are replaced by an edge from  $u$  to  $lca$ .

<sup>3</sup> We concentrate on a *multiple search tree* approach, where alternating trees are grown from all free vertices simultaneously. The *single search tree* approach, where just one alternating tree is grown at a time, is slightly simpler to describe, gives the same asymptotic running time, but leads to an inferior implementation.

<sup>4</sup> The crucial point is that any augmenting path in the resulting graph can be lifted to an augmenting path in the original graph.

However, we will regard these vertices to be conceptually shrunk into a new pseudo-vertex only. Since pseudo-vertices might get shrunk into other pseudo-vertices, the following view is appropriate: the current graph is partitioned into a *nested family* of odd cardinality subsets of  $V$ . Each odd cardinality subset is called a blossom. A blossom might contain other blossoms, called *subblossoms*. A *trivial* blossom corresponds to a single vertex of  $G$ . A blossom  $\mathcal{B}$  is said to be a *surface blossom*, if  $\mathcal{B}$  is not contained in another blossom. All edges lying completely in any blossom  $\mathcal{B}$  are *dead* and will not be considered by the algorithm; all other edges are *alive*.

*Dual Adjustment:* The algorithm might come to a halt due to the lack of further tight edges. Then, a *dual adjustment* is performed: the dual solution  $(y, z)$  is adjusted such that the objective value of  $(\overline{\text{WPM}})$  decreases. However, the adjustment will be of the kind such that  $(y, z)$  stays dual feasible and, moreover, preserves (CS)(1)–(2). One way to accomplish the desired result is to update the potentials  $y_u$  of all vertices  $u \in V$  and the potential  $z_{\mathcal{B}}$  of each non-trivial surface blossom  $\mathcal{B}$  by some  $\delta > 0$  as stated below:

$$y_u = y_u + \sigma\delta, \quad \text{and} \quad z_{\mathcal{B}} = z_{\mathcal{B}} - 2\sigma\delta.$$

The *status indicator*  $\sigma$  is defined as follows:  $\sigma$  equals  $-1$  or  $1$  for even or odd tree blossoms (trivial or non-trivial), respectively, and equals  $0$ , otherwise. The status of a vertex is the status of the surface blossom containing it. Let  $T_{r_1}, \dots, T_{r_k}$  denote the current alternating trees. The value of  $\delta$  must be chosen as  $\delta = \min\{\delta_2, \delta_3, \delta_4\}$ , with

$$\begin{aligned} \delta_2 &= \min_{uv \in E} \{\pi_{uv} : u^+ \in T_{r_i} \text{ and } v^\emptyset \text{ not in any tree}\}, \\ \delta_3 &= \min_{uv \in E} \{\pi_{uv}/2 : u^+ \in T_{r_i} \text{ and } v^+ \in T_{r_j}\}, \\ \delta_4 &= \min_{\mathcal{B} \in \mathcal{O}} \{z_{\mathcal{B}}/2 : \mathcal{B}^- \in T_{r_i}\}, \end{aligned}$$

where  $T_{r_i}$  and  $T_{r_j}$  denote any alternating tree, with  $1 \leq i, j \leq k$ .<sup>5</sup> The minimum of an empty set is defined to be  $\infty$ . When  $\delta = \infty$ , the dual linear program  $(\overline{\text{WPM}})$  is unbounded and thus no optimal solution to  $(\text{WPM})$  exists (by weak duality).

When  $\delta$  is chosen as  $\delta_4$ , the potential of an odd tree blossom, say  $\mathcal{B}^- \in T_r$ , will drop to  $0$  after the dual adjustment and therefore is not allowed to participate in further dual adjustments. The action to be taken is to *expand*  $\mathcal{B}$ , i.e., the defining subblossoms  $\mathcal{B}_1, \dots, \mathcal{B}_{2k+1}$  of  $\mathcal{B}$  are lifted to the surface and  $\mathcal{B}$  is abandoned. Since  $\mathcal{B}$  is odd, there exists a matching tree edge  $ub$  and a non-matching tree edge  $dv$ . The vertices  $b$  and  $d$  in  $\mathcal{B}$  are called the *base* and *discovery* vertex of  $\mathcal{B}$ . Assume  $\mathcal{B}_i$  and  $\mathcal{B}_j$  correspond to the subblossoms containing  $b$  and  $d$ , respectively. Let  $p$  denote the even length alternating (alive) path from  $\mathcal{B}_j$  to  $\mathcal{B}_i$  lying exclusively in  $\gamma(\mathcal{B})$ . The path  $p$  and therewith all subblossoms on  $p$  are added to  $T_r$ ; the subblossoms are labeled accordingly. All remaining subblossoms get unlabeled and leave  $T_r$ .

<sup>5</sup> Notice that  $T_{r_i}$  and  $T_{r_j}$  need not necessarily to be different in the definition of  $\delta_3$ .

*Realizations:* We argued before that the algorithm terminates after  $O(n)$  phases. The number of dual adjustments is bounded by  $O(n)$  per phase.<sup>6</sup> A *union-find* data structure supporting a *split* operation, additionally, is sufficient to maintain the surface graph in time  $O(m + n \log n)$  per phase.<sup>7</sup> The existing realizations of the blossom-shrinking algorithm differ in the way they determine the value of  $\delta$  and perform a dual adjustment.

The most trivial realization inspects all edges and explicitly updates the vertex and blossom potentials and thus needs  $O(n+m)$  time per dual adjustment. The resulting  $O(n^2m)$ , or  $O(n^4)$ , approach was suggested first by Edmonds [8].

Lawler [14] and Gabow [10] improved the asymptotic running time to  $O(n^3)$ . The idea is to keep the *best edge*, i.e., the edge having minimum reduced cost, for each non-tree vertex  $v^\circ$  and for each odd tree vertex  $v^- \in T_r$  to an even labeled tree vertex (the necessity of the latter is due to the expansion of blossoms). Moreover, each even tree blossom knows its best edges to other even tree blossoms. The time required for the determination of  $\delta$  and to perform a dual adjustment is therewith reduced to  $O(n)$ . The overall running time of  $O(n(m + n^2))$  ensues.

### 3 Implementation

*Blossom IV:* The implementation (called Blossom IV) of Cook and Rohe [4] is the most efficient code for weighted perfect matchings in general graphs currently available. The algorithm is implemented in C. The comparison to other implementations is made in two papers: (1) In [4] Blossom IV is compared to the implementation of Applegate and Cook [2]. It is shown that Blossom IV is substantially faster. (2) In [2] the implementation of Applegate and Cook is compared to other implementations. The authors show that their code is superior to all other codes.

A user of Blossom IV can choose between three modes: a single search tree approach, a multiple search tree approach, and a refinement of the multiple search tree approach, called the *variable  $\delta$  approach*. In the variable  $\delta$  approach, each alternating tree  $T_{r_i}$  chooses its own dual adjustment value  $\delta_{r_i}$  so as to maximize the decrease in the dual objective value. A heuristic is used to make the choices (an exact computation would be too costly). The variable  $\delta$  approach does not only lead to a faster decrease of the dual objective value, it, typically, also creates tight edges faster (at least, when the number of distinct edge weights is small). The experiments in [4] show that the variable  $\delta$  approach is superior to the other approaches in practice. We make all our comparisons to Blossom IV with the variable  $\delta$  approach.

<sup>6</sup> Whenever  $\delta = \delta_2, \delta_3$ , at least one vertex becomes an even labeled tree vertex, or a phase terminates. An even tree vertex stays even and resides in its tree until that tree gets destroyed. Thus,  $\delta = \delta_2, \delta_3$  may happen  $O(n)$  times per phase. The maximum cardinality of a blossom is  $n$  and thence  $\delta = \delta_4$  occurs  $O(n)$  times per phase.

<sup>7</sup> Each vertex knows the *name* of its surface blossom. On a *shrink* or an *expand* step all vertices of the smaller group are renamed.

Blossom IV uses a heuristic to find a good initial solution (jump start) and a price-and-repair heuristic for sparsening the input graph. We discuss both heuristics in Section 4.

*Our Implementation.* We come to our implementation. It has a worst-case running time of  $O(nm \log n)$ , uses (concatenable) priority queues extensively and is able to compute a non-perfect or a perfect maximum-weight matching. It is based on LEDA [15] and the implementation language is C++. The implementation can run either a single search tree approach or a multiple search tree approach; it turned out that the additional programming expenditure for the multiple search tree approach is well worth the effort regarding the efficiency in practice. Comparisons of our multiple search tree algorithm to the variable  $\delta$  approach of Blossom IV will be given in Section 4.

The underlying strategies are similar to or have been evolved from the ideas of Galil, Micali and Gabow [12]. However, our approach differs with regard to the maintenance of the varying potentials and reduced costs. Galil et al. handle these varying values within the priority queues, i.e., by means of an operation that changes all priorities in a priority queue by the same amount, whereas we establish a series of formulae that enable us to compute the values on demand. The time required to perform a dual adjustment is considerably improved to  $O(1)$ . Next, the key ideas of our implementation will be sketched briefly. As above, we concentrate on the description of the multiple search tree approach.

The definitions of  $\delta_2, \delta_3$  and  $\delta_4$  suggest to keep a priority queue for each of those; which we will denote by *delta2*, *delta3* and *delta4*, respectively. The priorities stored in each priority queue correspond to the value of interest, i.e., to (one half of) the reduced cost of edges for *delta2* and *delta3* and to blossom potentials for *delta4*. The minor difficulty that these priorities decrease by  $\delta$  with each dual adjustment is simply overcome as follows. We keep track of the amount  $\Delta = \sum \delta_i$  of all dual adjustments and compute the *actual* priority  $\tilde{p}$  of any element in the priority queues taking  $\Delta$  into consideration:  $\tilde{p} = p - \Delta$ . A dual adjustment by  $\delta$  then easily reduces to an increase of  $\Delta$  by  $\delta$ ; a dual adjustment takes time  $O(1)$  (additional details to affirm that will be given subsequently).

Some details for the maintenance of *delta2* are given next. We associate a *concatenable priority queue*  $P_{\mathcal{B}}$  with each surface blossom  $\mathcal{B}$ . A concatenable priority queue supports all standard priority queue operations and, in addition, a *concat* and *split* operation. Moreover, the elements are regarded to form a sequence. *concat* concatenates the sequences of two priority queues and, conversely, *split* splits the sequence of a priority queue at a given item into two priority queues. Both operations can be achieved to run in time  $O(\log n)$ , see, for example, [16, Section III.5.3] and [1, Section 4.12]. Our implementation is based on  $(2, 16)$ -trees.

$P_{\mathcal{B}}$  contains exactly one element  $\langle p, u \rangle$  for each vertex  $u$  contained in  $\mathcal{B}$ . Generally,  $p$  represents the reduced cost of the best edge of  $u$  to an even labeled tree vertex. More precise, let  $T_{r_1}, \dots, T_{r_k}$  denote the alternating trees at any stage of the blossom-shrinking approach. Every vertex  $u$  is associated with a series of (at most  $k$ ) incident edges  $uv_1, \dots, uv_k$  and their reduced costs  $\pi_{uv_1}, \dots, \pi_{uv_k}$

(maintained by a standard priority queue). Each edge  $uv_i$  represents the best edge from  $u$  to an even tree vertex  $v_i^+ \in T_{r_i}$ . The reduced cost  $\pi_{uv_i^*}$  of  $u$ 's best edge  $uv_i^*$  (along all best edges  $uv_i$  associated with  $u$ ), is the priority stored with the element  $\langle p, u \rangle$  in  $P_{\mathcal{B}}$ .

When a new blossom  $\mathcal{B}$  is formed by  $\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_{2k+1}$  the priority queues  $P_{\mathcal{B}_1}, P_{\mathcal{B}_2}, \dots, P_{\mathcal{B}_{2k+1}}$  are concatenated one after another and the resulting priority queue  $P_{\mathcal{B}}$  is assigned to  $\mathcal{B}$ . Thereby, we keep track of each  $t_i$ ,  $1 \leq i \leq 2k+1$ , the last item in  $\mathcal{B}_i$ . Later, when  $\mathcal{B}$  gets expanded the priority queues to  $\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_{2k+1}$  can easily be recovered by splitting the priority queue of  $\mathcal{B}$  at each item  $t_i$ ,  $1 \leq i \leq 2k+1$ .

Whenever a blossom  $\mathcal{B}$  becomes unlabeled, it sends its best edge and the reduced cost of that edge to *delta2*. Moreover, when the best edge of  $\mathcal{B}$  changes, the appropriate element in *delta2* is adjusted. When a non-tree blossom  $\mathcal{B}^\emptyset$  becomes an odd tree blossom its element in *delta2* is deleted. These insertions, adjustments and deletions on *delta2* contribute  $O(n \log n)$  time per phase.

We come to *delta3*. Each tree  $T_{r_i}$  maintains its own priority queue  $\text{delta3}_{r_i}$  containing all *blossom forming* edges, i.e., the edges connecting two even vertices in  $T_{r_i}$ . The priority of each element corresponds to one half of the reduced cost; note, however, that the actual reduced cost of each element is computed as stated above. The edges inserted into  $\text{delta3}_{r_i}$  are assured to be alive. However, during the course of the algorithm some edges in  $\text{delta3}_{r_i}$  might become dead. We use a *lazy-deletion* strategy for these edges: dead edges are simply discarded when they occur as the minimum element of  $\text{delta3}_{r_i}$ . The minimum element of each  $\text{delta3}_{r_i}$  is sent to *delta3*. Moreover, each tree  $T_{r_i}$  sends its best edge  $uv$  with  $u^+ \in T_{r_i}$  and  $v^+ \in T_{r_j}$ ,  $T_{r_i} \neq T_{r_j}$ , to *delta3*. When a tree  $T_{r_i}$  gets destroyed, its (two) representatives are deleted from *delta3* and  $\text{delta3}_{r_i}$  is freed. Since  $m \leq n^2$ , the time needed for the maintenance of all priority queues responsible for *delta3* is  $O(m \log n)$  per phase.

Handling *delta4* is trivial. Each non-trivial odd surface blossom  $\mathcal{B}^- \in T_{r_i}$  sends an element to *delta4*. The priority corresponds to one half of the potential  $z_{\mathcal{B}}$ .

What remains to be shown is how to treat the varying blossom and vertex potentials as well as the reduced cost of all edges associated with the vertices. The crux is that with each dual adjustment all these values uniformly change by some amount of  $\delta$ .

For example, consider an even surface blossom  $\mathcal{B}^+ \in T_{r_i}$ . The potential of  $\mathcal{B}$  changes by  $+2\delta$ , the potential of each vertex  $u \in \mathcal{B}$  by  $-\delta$  and the reduced costs of all edges associated with each  $u \in \mathcal{B}$  change by  $-2\delta$  with a dual adjustment by  $\delta$ . Taking advantage of that fact, the actual value of interest can again be computed by taking  $\Delta$  and some additional information into consideration. The idea is as follows.

Each surface blossom  $\mathcal{B}$  has an offset  $\text{offset}_{\mathcal{B}}$  assigned to it. This offset is initially set to 0 and will be adjusted whenever  $\mathcal{B}$  changes its status. The formulae to compute the actual potential  $\tilde{z}_{\mathcal{B}}$  for a (non-trivial) surface blossom  $\mathcal{B}$ , the actual potential  $\tilde{y}_u$  for a vertex  $u$  (with surface blossom  $\mathcal{B}$ ) and the actual reduced

cost  $\tilde{\pi}_{uv_i}$  of an edge  $uv_i$  associated with  $u$  (and surface blossom  $\mathcal{B}$ ) are given below:

$$\tilde{z}_{\mathcal{B}} = z_{\mathcal{B}} - 2\text{offset}_{\mathcal{B}} - 2\sigma\Delta, \quad (1)$$

$$\tilde{y}_u = y_u + \text{offset}_{\mathcal{B}} + \sigma\Delta, \quad (2)$$

$$\tilde{\pi}_{uv_i} = \pi_{uv_i} + \text{offset}_{\mathcal{B}} + (\sigma - 1)\Delta. \quad (3)$$

Here,  $\sigma$  and  $\Delta$  are defined as above. It is not difficult to affirm the offset update of a surface blossom  $\mathcal{B}$ :

$$\text{offset}_{\mathcal{B}} = \text{offset}_{\mathcal{B}} + (\sigma - \sigma')\Delta, \quad (4)$$

which is necessary at the point of time, when  $\mathcal{B}$  changes its status indicator from  $\sigma$  to  $\sigma'$ . To update the blossom offset takes time  $O(1)$ . We conclude, that each value of interest can be computed (if required) in time  $O(1)$  by the formulae (1)–(3).

It is an easy matter to handle the blossom offsets in an expand step for  $\mathcal{B}$ :  $\text{offset}_{\mathcal{B}}$  is assigned to each offset of the defining subblossoms of  $\mathcal{B}$ . However, it is not obvious in which way one can cope with different blossom offsets in a shrink step. Let  $\mathcal{B}$  denote the blossom that is going to be formed by the defining subblossoms  $\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_{2k+1}$ . Each odd labeled subblossom  $\mathcal{B}_i$  is made even by adjusting its offset as in (4). The corresponding offsets  $\text{offset}_{\mathcal{B}_1}, \text{offset}_{\mathcal{B}_2}, \dots, \text{offset}_{\mathcal{B}_{2k+1}}$  may differ in value. However, we want to determine a common offset value  $\text{offset}_{\mathcal{B}}$  such that the actual potential and the actual reduced costs associated with each vertex  $u \in \mathcal{B}_i$  can be computed with respect to  $\text{offset}_{\mathcal{B}}$ .

The following strategy assures that the offsets of all defining subblossoms  $\mathcal{B}_i$ ,  $1 \leq i \leq 2k+1$ , are set to zero and thus the desired result is achieved by the common offset  $\text{offset}_{\mathcal{B}} = 0$ .

Whenever a surface blossom  $\mathcal{B}'$  (trivial or non-trivial) becomes an even tree blossom, its offset  $\text{offset}_{\mathcal{B}'}$  is set to zero. Consequently, in order to preserve the validity of (2) and (1) for the computation of the actual potential  $\tilde{y}_u$  of each vertex  $u \in \mathcal{B}'$  and of the actual potential  $\tilde{z}_{\mathcal{B}'}$  of  $\mathcal{B}'$  itself (when  $\mathcal{B}'$  is non-trivial only), the following adjustments have to be performed:

$$\begin{aligned} y_u &= y_u + \text{offset}_{\mathcal{B}'}, \\ z_{\mathcal{B}'} &= z_{\mathcal{B}'} - 2\text{offset}_{\mathcal{B}'}. \end{aligned}$$

Moreover, the stored reduced cost  $\pi_{uv_i}$  of each edge  $uv_i$  associated with each vertex  $u \in \mathcal{B}'$  is subject to correction:

$$\pi_{uv_i} = \pi_{uv_i} + \text{offset}_{\mathcal{B}'}.$$

Observe that the adjustments are performed at most once per phase for a fixed vertex. Thus, the time required for the potential adjustments is  $O(n)$  per phase. On the other hand, the corrections of the reduced costs contributes total time  $O(m \log n)$  per phase.



In summary, we have established a convenient way to handle the varying potentials as well as the reduced costs of edges associated with a vertex. The values of interest can be computed on demand by the formulae developed. The additional overhead produced by the offset maintenance has been proved to consume  $O(m \log n)$  time per phase. This concludes the description of our approach (for a more extensive discussion the reader is referred to [17]).

*Correctness:* Blossom IV and our program does not only compute an optimal matching  $M$  but also an optimal dual solution. This makes it easy to verify the correctness of a solution. One only has to check that  $M$  is a (perfect) matching, that the dual solution is feasible (in particular, all edges must have non-negative reduced costs), and that the complementary slackness conditions are satisfied.

## 4 Experimental Results

We experimented with three kinds of instances: Delaunay instances, (sparse and dense) random instances having a perfect matching, and complete geometric instances.

For the Delaunay instances we chose  $n$  random points in the unit square and computed their Delaunay triangulation using the LEDA Delaunay implementation. The edge weights correspond to the Euclidean distances scaled to integers in the range  $[1, \dots, 2^{16})$ . Delaunay graphs are known to contain perfect matchings [7].

For the random instances we chose random graphs with  $n$  vertices. The number of edges for sparse graphs was chosen as  $m = \alpha n$  for small values of  $\alpha$ ,  $\alpha \leq 10$ . For dense graphs the density is approximately 20%, 40% and 60% of the density of a complete graph. Random weights out of the range  $[1, \dots, 2^{16})$  were assigned to the edges. We checked for perfect matchings with the LEDA cardinality matching implementation.

Complete geometric instances were induced by  $n$  random points in an  $n \times n$  square and their Euclidean distances.

*Experimental Setting:* All our running times are in seconds and are the average of  $t = 5$  runs, unless stated otherwise. All experiments were performed on a Sun Ultra Sparc, 333 Mhz.

*Initial Solution, Single and Multiple Search Tree Strategies:* We implemented two strategies for finding initial solutions, both of them well known and also used in previous codes. The *greedy heuristics* first sets the vertex potentials: the potential  $y_v$  of a vertex  $v$  is set to one-half the weight of the heaviest incident edge. This guarantees that all edges have non-negative reduced cost. It then chooses a matching within the tight edges in a greedy fashion. The fractional matching heuristic [5] first solves the fractional matching problem (constraints (WPM)(1) and (WPM)(3)). In the solution all variables are half-integral and the edges with value  $1/2$  form odd length cycles. The initial matching consists of the

edges with value 1 and of  $\lfloor |C|/2 \rfloor$  edges from every odd cycle. Applegate and Cook [2] describe how to solve the fractional matching problem. Our fractional matching algorithm uses priority queues and similar strategies to the one evolved above. The priority queue based approach appears to be highly efficient.<sup>8</sup> Table 1 compares the usage of different heuristics in combination with the single search tree (SST) and the multiple search tree (MST) approach.

**Table 1.** SST vs. MST on Delaunay graphs.  $-$ ,  $+$  or  $*$  indicates usage of no, the greedy or the fractional matching heuristic. The time needed by the greedy or the fractional matching heuristic is shown in the columns GY and FM.

$n$	SST $^-$	MST $^-$	SST $^+$	MST $^+$	GY	SST $^*$	MST $^*$	FM	$t$
10000	37.01	6.27	24.05	4.91	0.13	5.79	3.20	0.40	5
20000	142.93	14.81	89.55	11.67	0.24	18.54	8.00	0.83	5
40000	593.58	31.53	367.37	25.51	0.64	76.73	17.41	1.78	5

The fractional matching heuristic is computational more intensive than the greedy heuristic, but leads to overall improvements of the running time. The multiple search tree strategy is superior to the single search tree strategy with both heuristics. We therefore take the multiple search tree strategy with the fractional matching heuristic as our canonical implementation; we refer to this implementation as MST\*. Blossom IV also uses the fractional matching heuristic for constructing an initial solution. We remark that the difference between the two heuristics is more pronounced for the single search tree approach.

Table 2 compares Blossom IV with the fractional matching heuristic, multiple search trees without (B4\*) and with (B4\*<sub>var</sub>) variable  $\delta$ 's with MST\*. We used Delaunay graphs.

**Table 2.** B4\*, B4\*<sub>var</sub> vs. MST\* on Delaunay graphs.

$n$	B4*	B4* <sub>var</sub>	MST*	$t$
10000	73.57	4.11	3.37	5
20000	282.20	12.34	7.36	5
40000	1176.58	29.76	15.84	5

The table shows that the variable  $\delta$  approach B4\*<sub>var</sub> makes a tremendous difference for Blossom IV and that MST\* is competitive with B4\*<sub>var</sub>.

*Influence of Edge Weights:* Table 3 shows the influence of the edge weights on the running time. We took random graphs with  $m = 4n$  edges and random edge weights in the range  $[1, \dots, b]$  for different values of  $b$ . For  $b = 1$ , the problem is unweighted.

<sup>8</sup> Experimental results (not presented in this paper) showed that the priority queue approach is substantially faster than the specialized algorithm of LEDA to compute a maximum-weight (perfect) matching in a bipartite graph.

**Table 3.**  $B4^*$ ,  $B4_{\text{var}}^*$  vs.  $MST^*$  on random graphs with variable weight range.

$n$	$\alpha$	$b$	$B4^*$	$B4_{\text{var}}^*$	$MST^*$	$t$
10000	4	1	3.98	3.99	0.85	1
10000	4	10	2.49	3.03	2.31	1
10000	4	100	3.09	3.10	2.58	1
10000	4	1000	17.41	8.40	2.91	1
10000	4	10000	13.69	11.91	2.78	1
10000	4	100000	12.06	11.20	2.69	1

The running time of  $B4^*$  and  $B4_{\text{var}}^*$  depends significantly on the size of the range, the running time of  $MST^*$  depends only weakly (except for the unweighted case which is simpler).  $MST^*$  is superior to  $B4_{\text{var}}^*$ .

We try an explanation. When the range of edge weights is small, a dual adjustment is more likely to make more than one edge tight. Also it seems to take fewer dual adjustments until an augmenting path is found. Since dual adjustments are cheaper in our implementation ( $O(m \log n)$  for all adjustments in a phase of our implementation versus  $O(n)$  for a single adjustment in Blossom IV), our implementation is less harmed by large numbers of adjustments.

*Asymptotics:* Tables 4 and 5 give some information about the asymptotics. For Tab. 4 we have fixed the graph density at  $m = 6n$  and varied  $n$ .

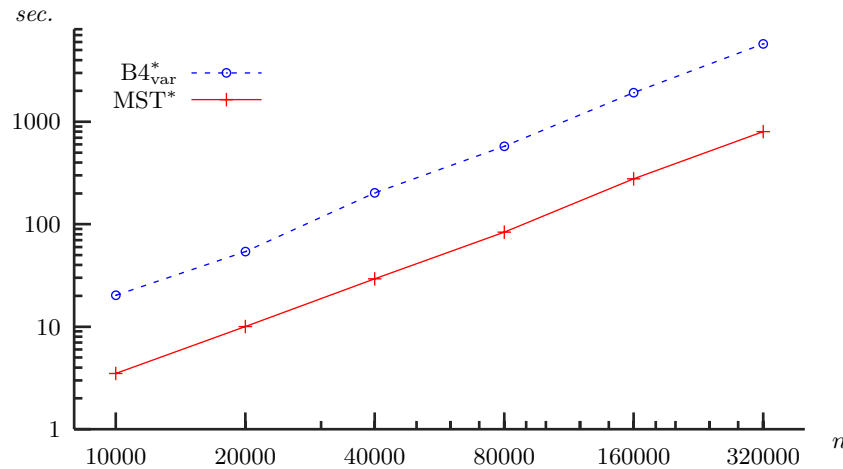
**Table 4.**  $B4^*$ ,  $B4_{\text{var}}^*$  vs.  $MST^*$  on sparse random graphs with  $m = 6n$ .

$n$	$\alpha$	$B4^*$	$B4_{\text{var}}^*$	$MST^*$	$t$
10000	6	20.94	18.03	3.51	5
20000	6	82.96	53.87	9.97	5
40000	6	194.48	177.28	29.05	5

The running times of  $B4_{\text{var}}^*$  and  $MST^*$  seem to grow less than quadratically (with  $B4_{\text{var}}^*$  taking about six times as long as  $MST^*$ ). Table 5 gives more detailed information. We varied  $n$  and  $\alpha$ .

**Table 5.**  $B4^*$ ,  $B4_{\text{var}}^*$  vs.  $MST^*$  on sparse random graphs with  $m = \alpha n$ .

$n$	$\alpha$	$B4^*$	$B4_{\text{var}}^*$	$MST^*$	$t$
10000	6	20.90	20.22	3.49	5
10000	8	48.50	22.83	5.18	5
10000	10	37.49	30.78	5.41	5
20000	6	96.34	54.08	10.04	5
20000	8	175.55	89.75	12.20	5
20000	10	264.80	102.53	15.06	5
40000	6	209.84	202.51	29.27	5
40000	8	250.51	249.83	36.18	5
40000	10	710.08	310.76	46.57	5



**Fig. 1.** Asymptotics of  $B4_{\text{var}}^*$  and  $MST^*$  algorithm on sparse random instances ( $\alpha = 6$ ).

A log-log plot indicating the asymptotics of  $B4_{\text{var}}^*$  and our  $MST^*$  algorithm on random instances ( $\alpha = 6$ ) is depicted in Fig. 1.

*Variance in Running Time:* Table 6 gives information about the variance in running time. We give the best, worst, and average time of five instances. The fluctuation is about the same for both implementations.

**Table 6.**  $B4_{\text{var}}^*$  vs.  $MST^*$  on sparse random graphs with  $m = 6n$ .

$n$	$\alpha$	$B4_{\text{var}}^*$			$MST^*$			$t$
		best	worst	average	best	worst	average	
10000	6	16.88	20.03	18.83	3.34	4.22	3.78	5
20000	6	49.02	60.74	55.15	9.93	11.09	10.30	5
40000	6	162.91	198.11	180.88	25.13	32.24	29.09	5

*Dense Graphs and Price and Repair:* Our experiments suggest that  $MST^*$  is superior to  $B4_{\text{var}}^*$  on sparse graphs. Table 7 shows the running time on dense graphs. Our algorithm is superior to Blossom IV even on these instances.

Blossom IV provides a price-and-repair heuristic which allows it to run on implicitly defined complete geometric graphs (edge weight = Euclidean distance). The running time on these instances is significantly improved for  $B4_{\text{var}}^*$  using the price-and-repair heuristic as can be seen in Tab. 8. We have not yet implemented such a heuristic for our algorithm.

The idea is simple. A minimum-weight matching (it is now more natural to talk about minimum-weight matchings) has a natural tendency of avoiding large weight edges; this suggests to compute a minimum-weight matching iteratively. One starts with a sparse subgraph of light edges and computes an optimal

**Table 7.**  $B4_{\text{var}}^*$  vs.  $MST^*$  on dense random graphs. The density is approximately 20%, 40% and 60%.

$n$	$m$	$B4^*$	$B4_{\text{var}}^*$	$MST^*$	$t$
1000	100000	6.97	5.84	1.76	5
1000	200000	16.61	11.35	3.88	5
1000	300000	18.91	18.88	5.79	5
2000	200000	46.71	38.86	8.69	5
2000	400000	70.52	70.13	16.37	5
2000	600000	118.07	115.66	23.46	5
4000	400000	233.16	229.51	42.32	5
4000	800000	473.51	410.43	92.55	5
4000	1200000	523.40	522.52	157.00	5

**Table 8.** Blossom IV variable  $\delta$  without ( $B4_{\text{var}}^*$ ) and with ( $B4_{\text{var}}^{**}$ ) price-and-repair heuristic vs.  $MST^*$  on complete geometric instances. The Delaunay graph of the point set was chosen as the sparse subgraph for  $B4_{\text{var}}^{**}$ .

$n$	$B4_{\text{var}}^*$	$B4_{\text{var}}^{**}$	$MST^*$	$t$
1000	37.01	0.43	24.05	5
2000	225.93	1.10	104.51	5
4000	1789.44	4.33	548.19	5

matching. Once the optimal matching is computed, one checks optimality with respect to the full graph. Any edge of negative reduced cost is added to the graph and primal and dual solution are modified so as to satisfy the preconditions of the matching algorithm. Derigs and Metz [6,2,4] discuss the repair step in detail.

There are several natural strategies for selecting the sparse subgraph. One can, for example, take the lightest  $d$  edges incident to any vertex. For complete graphs induced by a set of points in the plane and with edge weights equal to the Euclidean distance, the Delaunay diagram of the points is a good choice.

*‘Worse-case’ Instances for Blossom IV:* We wish to conclude the experiments with two ‘worse-case’ instances that demonstrate the superiority of our algorithm to Blossom IV.

The first ‘worse-case’ instance for Blossom IV is simply a chain. We constructed a chain having  $2n$  vertices and  $2n - 1$  edges. The edge weights along the chain were alternately set to 0 and 2 (the edge weight of the first and last edge equal 0).  $B4_{\text{var}}^*$  and our  $MST^*$  algorithm were asked to compute a maximum-weight perfect matching. Note that the fractional matching heuristic will always compute an optimal solution on instances of this kind. Table 9 shows the results.

The running time of  $B4_{\text{var}}^*$  grows more than quadratically (as a function of  $n$ ), whereas the running time of our  $MST^*$  algorithm grows about linearly with  $n$ . We present our argument as to why this is to be expected. First of all, the greedy heuristic will match all edges having weight 2; the two outer vertices remain unmatched. Each algorithm will then have to perform  $O(n)$  dual adjustments so as to obtain the optimal matching. A dual adjustment takes time

**Table 9.**  $B4^*_{\text{var}}$  vs.  $MST^*$  on chains.

$2n$	$B4^*_{\text{var}}$	$MST^*$	$t$
10000	94.75	0.25	1
20000	466.86	0.64	1
40000	2151.33	2.08	1

$O(n)$  for Blossom IV (each potential is explicitly updated), whereas it takes  $O(1)$  for our  $MST^*$  algorithm. Thus, Blossom IV will need time  $O(n^2)$  for all these adjustments and, on the other hand, the time required by our  $MST^*$  algorithm will be  $O(n)$ .

Another ‘worse-case’ instance for Blossom IV occurred in VLSI-Design having  $n = 151780$  vertices and  $m = 881317$  edges. Kindly, Andreas Rohe made this instance available to us. We compared  $B4^*$  and  $B4^*_{\text{var}}$  to our  $MST$  algorithm. We ran our algorithm with the greedy heuristic ( $MST^+$ ) as well as with the fractional matching heuristic ( $MST^*$ ). The results are given in Tab. 10.

**Table 10.**  $B4^*$ ,  $B4^*_{\text{var}}$  vs.  $MST$  on `boese.edg` instance.

$n$	$m$	$B4^*$	$B4^*_{\text{var}}$	$MST^+$	$MST^*$	$t$
151780	881317	200019.74	200810.35	3172.70	5993.61	1
		(332.01)	(350.18)	(5.66)	(3030.35)	

The second row states the times that were needed by the heuristics. Observe that both Blossom IV algorithms need more than two days to compute an optimal matching, whereas our algorithm solves the same instance in less than an hour. For our  $MST$  algorithm the fractional matching heuristic did not help at all on this instance: to compute a fractional matching took almost as long as computing an optimum matching for the original graph (using the greedy heuristic).

## 5 Conclusion

We described the implementation of an  $O(nm \log n)$  matching algorithm. Our implementation is competitive to the most efficient known implementation due to Cook and Rohe [4]. Our research rises several questions. (1) Is it possible to integrate the variable  $\delta$  approach into an  $O(nm \log n)$  algorithm? (2) A generator of instances forcing the implementation into their worst-case would be useful. (3) In order to handle complete graphs more efficiently the effect of a price-and-repair strategy is worth being considered; most likely, providing such a mechanism for  $MST^*$  will improve its worst-case behaviour on those graphs tremendously. (4) Internally, the underlying graph is represented by the leda class *graph*— which allows for dynamic operations like *new\_node*, etc. However, dynamic operations are not required by our algorithm. The running time of some

other graph algorithms in LEDA improved by a factor of about two using a static variant of the *graph* data structure (Stefan Näher: personal communication). Probably, a similar effect can be achieved for our maximum-weight matching algorithm as well.

## References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullmann. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.
2. D. Applegate and W. Cook. Solving large-scale matching problems. In D. Johnson and C.C. McGeoch, editors, *Network Flows and Matchings*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 557–576. American Mathematical Society, 1993.
3. B. Cherkassky and A. Goldberg. PRF, a Maxflow Code. [www.intertrust.com/star/goldberg/index.html](http://www.intertrust.com/star/goldberg/index.html).
4. W. Cook and A. Rohe. Computing minimum-weight perfect matchings. Technical Report 97863, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, 1997.
5. U. Derigs and A. Metz. On the use of optimal fractional matchings for solving the (integer) matching problem. *Mathematical Programming*, 36:263–270, 1986.
6. U. Derigs and A. Metz. Solving (large scale) matching problems combinatorially. *Mathematical Programming*, 50:113–122, 1991.
7. M.B. Dillencourt. Toughness and delaunay triangulations. *Discrete and Computational Geometry*, 5:575–601, 1990.
8. J. Edmonds. Maximum matching and a polyhedron with  $(0,1)$  vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.
9. J. Edmonds. Paths, trees, and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.
10. H. N. Gabow. *Implementation of algorithms for maximum matching and nonbipartite graphs*. PhD thesis, Stanford University, 1974.
11. H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In David Johnson, editor, *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 434–443, San Francisco, CA, USA, January 1990. SIAM.
12. Z. Galil, S. Micali, and H. N. Gabow. An  $O(EV \log V)$  algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Computing*, 15:120–130, 1986.
13. M. Humble. Implementierung von Flußalgorithmen mit dynamischen Bäumen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1996.
14. E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.
15. K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.
16. K. Mehlhorn. *Data structures and algorithms. Volume 1: Sorting and searching*, volume 1 of *EATCS monographs on theoretical computer science*. Springer, 1984.
17. G. Schäfer. Weighted matchings in general graphs. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2000.

# Pushing the Limits in Sequential Sorting

Stefan Edelkamp and Patrick Stiegeler

Institut für Informatik, Albert-Ludwigs-Universität,  
Georges-Köhler-Allee, D-79110 Freiburg  
{edelkamp,stiegele}@informatik.uni-freiburg.de

**Abstract.** With refinements to the *WEAK-HEAPSORT* algorithm we establish the general and practical relevant sequential sorting algorithm *RELAXED-WEAK-HEAPSORT* executing exactly  $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + 1 \leq n\log n - 0.9n$  comparisons on any given input. The number of transpositions is bounded by  $n$  plus the number of comparisons. Experiments show that *RELAXED-WEAK-HEAPSORT* only requires  $O(n)$  extra bits. Even if this space is not available, with *QUICK-WEAK-HEAPSORT* we propose an efficient *QUICKSORT* variant with  $n\log n + 0.2n + o(n)$  comparisons on the average. Furthermore, we present data showing that *WEAK-HEAPSORT*, *RELAXED-WEAK-HEAPSORT* and *QUICK-WEAK-HEAPSORT* beat other performant *QUICKSORT* and *HEAPSORT* variants even for moderate values of  $n$ .

## 1 Introduction

Similar to *Fibonacci-Heaps* (Fredman and Tarjan (1987)), *Weak-Heaps* introduced by Dutton (1992) are obtained by relaxing the heap requirements. More precisely, a *Weak-Heap* is a binary tree representation of a totally ordered set, satisfying the following three conditions: The root value of any subtree is larger than or equal to all elements to its right, the root of the entire heap structure has no left child, and leaves are found on the last two levels only.

The array representation utilizes so-called *reverse* bits. The index of the left child is located at  $2i + \text{Reverse}[i]$  and the right child is found at  $2i + 1 - \text{Reverse}[i]$ . For this purpose  $\text{Reverse}[i]$  is interpreted as an integer in  $\{0, 1\}$ , being initialized with value 0. Therefore, by flipping  $\text{Reverse}[i]$  the indices of the left and the right child are exchanged, which simulates a rotation of the subtrees at  $i$ . Edelkamp and Wegener (2000) prove that the worst case number of comparisons in the *WEAK-HEAPSORT* algorithm (Dutton (1993)) is bounded by  $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + n - \lceil\log n\rceil$ , show that the best case is given by  $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + 1$  comparisons and report experiments on the average case with  $n\log n + d(n)n$  comparisons and  $d(n) \in [-0.47, -0.42]$ . Moreover, the authors present examples matching the worst and the best case bounds, determine the number of *Weak-Heaps* on  $\{1, \dots, n\}$ , and suggest a double-ended priority queue to be generated in optimal  $n + \lceil n/2 \rceil - 2$  comparisons.

We assume that all elements are kept in an array structure and distinguish two scenarios: In the first case of *external sorting*, we assume that we have access



to  $O(n)$  extra space for the output data and in the second case of *internal sorting* the permutation  $\pi \in S_n$  with  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$  is implicitly given by committed moves of elements. The term *incrementally external* reflects that the output data is exclusively written.

We may also build an index for the permutation  $\pi \in S_n$ . In this case transpositions and assignments are just pointer manipulations, but key comparisons still involve access to the data. Therefore, the number of comparisons dominate the time complexity. However, since the index to express  $\pi$  requires at least  $O(n \log n)$  bits, we say that the *In-Place (In-Situ)* property is violated. Note that an additional Boolean array *Active* may transform any *incrementally external* sorting algorithm into an *index-based* implementation. Each element to be flushed is unmarked and its final position is stored in the index. Therefore, at most  $2n$  further assignments finally sort the element array according to  $\pi$ .

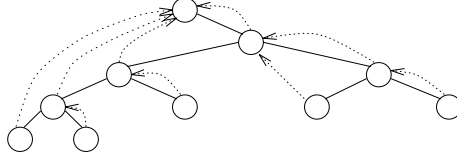
In case of simple keys the representation length is bounded by a small constant. Special-tailored algorithms like *RADIXSORT* (Nilsson (1996)) exhibit this structure for a performance even beyond the lower bound of  $\lceil \log(n!) \rceil \approx n \log n - 1.4427n$  comparisons in the worst and  $\lceil \log(n!) \rceil - 1$  comparisons in the average case. Our focus, however, will be general sequential sorting, which applies to any totally ordered set.

This paper is structured as follows: First we briefly sketch the *WEAK-HEAPSORT* algorithm and its properties. Then we address refinements to the algorithm yielding the main idea for the incrementally external variant *EXTERNAL-WEAK-HEAPSORT*: The relaxed placing scheme. Subsequently, we show how to implement the *index-based* version *RELAXED-WEAK-HEAPSORT* and focus on a space-saving strategy that achieves the claimed bound on  $O(n)$  bits in the experiments. Then we will turn to the *QUICK-WEAK-HEAPSORT*, which in fact is a performant extension to the *QUICK-HEAPSORT* algorithm recently proposed by Cantone and Cincotti (2000). Both algorithms are implemented to compete with various performant *QUICKSORT* and *HEAPSORT* variants. Finally we give some concluding remarks.

## 2 The *WEAK-HEAPSORT* Algorithm

Analogous to the *HEAPSORT* algorithm *WEAK-HEAPSORT* can be partitioned into two phases: *heap-creation* and *sorting*. In the latter phase we successively extract the root element and readjust the remaining *Weak-Heap* structure. For *heap creation* we consider the following recursively defined *grandparent* relationship:  $Gparent(x) = Gparent(Parent(x))$ , if  $x$  is a left child, and  $Gparent(x) = Parent(x)$ , if  $x$  is a right child (Fig. 1). In a *Weak-Heap*,  $Gparent(i)$  is the index of the lowest element known to be bigger than or equal to the one at  $i$ .

Let  $\langle x, T_1 \rangle$  and  $\langle y, T_2 \rangle$  be two *Weak-Heaps* with  $rchild(x) = root(T_1)$  and  $rchild(y) = root(T_2)$ . We assume  $a_x \geq a_y$  and that the leaves of  $T_1$  and  $T_2$  have depth values that differ by at most one. Then the following *Merge'* sequence leads to a new *Weak-Heap*  $T$ :  $root(T)$  is set to  $x$ ,  $lchild(y)$  is set to  $rchild(x)$ ,



**Fig. 1.** The grandparent relationship  $Gparent$ .

```

PROCEDURE Gparent(j)
  WHILE (j MOD 2 = 0)
    j := j DIV 2
  RETURN j DIV 2

PROCEDURE Merge(i, j)
  IF (a[i] < a[j])
    Swap(i, j)
    Reverse[j] := NOT Reverse[j]

PROCEDURE WeakHeapify
  FOR j := n-1 DOWNT0 1
    Merge(Gparent(j), j)

PROCEDURE MergeForest(m)
  x := 1
  WHILE (2x + Reverse[x] < m)
    x := 2x + Reverse[x]
  WHILE (x > 0)
    Merge(m, x)
    x := x DIV 2

PROCEDURE WeakHeapSort
  WeakHeapify
  a[n] := a[0]
  FOR i := n-1 DOWNT0 2
    MergeForest(i)
  FOR i := 0 TO n-1
    a[i] := a[i+1]

```

**Fig. 2.** The implementation of the *WEAK-HEAPSORT* algorithm. Divisions and multiplications with 2 are realized with Boolean shift operations.

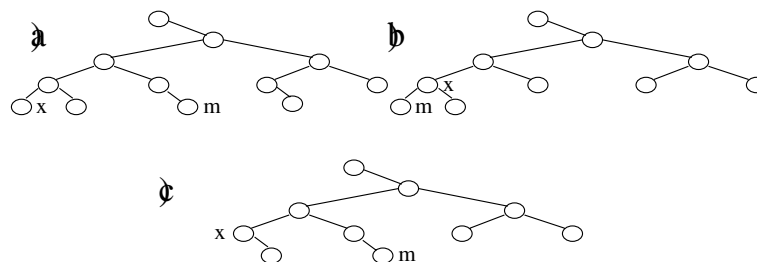
and  $rchild(x)$  is set to  $y$ . The modified *Merge* function operates on a node  $x$  and a binary tree  $T = (lT(y), y, rT(y))$  as follows. If  $\langle x, lT(y) \rangle, \langle y, rT(y) \rangle$  are *Weak-Heaps*, then  $Merge(x, T) = Merge'(\langle x, lT(y) \rangle, \langle y, rT(y) \rangle)$  leads to a correct *Weak-Heap* according to its definition. We exhibit two cases. If  $a_x \geq a_y$  then  $x$  is assigned to the root of the new tree with  $rchild(x)$  being set to  $y$ . If  $a_y > a_x$  then  $y$  is assigned the root of the new tree with  $lchild(x)$  being set to  $rchild(y)$ ,  $rchild(x)$  being set to  $lchild(y)$  and  $rchild(y)$  being set to  $x$ . Hence, the subtrees with respect to  $lchild(x)$  and  $rchild(y)$  are rotated.

For initializing an unsorted binary tree  $T$  with  $WeakHeapify(T)$  in a bottom-up traversal every subtree rooted at a node  $j$  is merged to its grandparent  $Gparent(j)$ . Therefore,  $Merge(Gparent(j), j)$  is invoked only if  $Merge(Gparent(rchild(j)), rchild(j))$  and  $Merge(Gparent(lchild(j)), lchild(j))$  have terminated finally yielding a correct *Weak-Heap* in  $n - 1$  comparisons.

Let the *grandchildren relation*  $S_x$  be defined as the inverse to the *grandparent relation*, i.e.  $y \in S_x$  iff  $Gparent(y) = x$ . To retain a correct *Weak-Heap* structure after the extraction of the root element, the operation *MergeForest* performs a bottom-up traversal of the *special path*  $S_{root}(T)$ . By the definition of *Weak-Heaps*, the second largest element in  $T$  is contained in  $S_{root}(T)$ . Let  $m$  be a freely chosen element on the lowest level of the remaining structure representing the next root element. Until the root node is reached in the operation *MergeForest*, we iterate on the next two operations:  $Merge(m, x)$  and  $x \leftarrow Parent(x)$ , with  $x$  representing the subtrees  $T_x$  for  $x \in S_{root}(T)$ . Since  $\langle x, T_x \rangle, x \in S_{root}(T)$ , are all *Weak-Heaps* the operation *MergeForest* leads to a correct *Weak-Heap* according to its definition. Fig. 2 provides the pseudo-code of *WEAK-HEAPSORT*.

### 3 Refinements to the Algorithm

The presented *WEAK-HEAPSORT* algorithm of Dutton can be improved leading to an algorithm that always achieves the best case number of comparisons. To understand the difference between the worst and the best case consider the different cases that may occur while executing the algorithm (cf. Fig. 3).



**Fig. 3.** Three different cases that may occur in the *MergeForest* operation.

Case a) is the situation where  $x$  is placed on the bottommost level. The other cases are b), with position  $m$  being located on the special path, and c), otherwise. In a)  $\lceil \log(m+1) \rceil$  and in b) and c)  $\lceil \log(m+1) \rceil - 1$  comparisons are required. Evaluating  $n - 1 + \sum_{m=2}^{n-1} (\lceil \log(m+1) \rceil - 1)$  leads to the following result.

**Theorem 1.** *Let  $k = \lceil \log n \rceil$ . The best case number of comparisons in *WEAK-HEAPSORT* equals  $nk - 2^k + 1 \leq n \log n - 0.913987n$ . The number of assignments is  $n + 1$  and the number of transpositions is bounded by the number of comparisons.*

Analogously, the number of comparisons is bounded by  $nk - 2^k + n - k \leq n \log n + 0.086013n$ . These bounds cannot be improved upon, since Edelkamp and Wegener (2000) provide scalable exactly matching worst case and best case examples. Therefore, in order to get better bounds, the *WEAK-HEAPSORT* algorithm itself has to be changed.

#### 3.1 Next-to- $m$ Strategy

The potential on savings is exemplified with a simple strategy gaining about  $k$  comparisons with respect to the original algorithm. The implementation is depicted in Fig. 4. (Note that, the depth of  $x$  and  $m$  are equal if the number  $x$  AND  $m$  is greater than  $x$  XOR  $m$ .) The first condition guarantees that case a) is met, in which the worst case number of  $\lceil \log(m+1) \rceil$  comparison is given. In this case the situation  $m - 1 = x$  is distilled such that the element at  $x$  is located next to the element at  $m$ . The next root node is  $m - 1$  such that the special path is shortened by one. After the merging process the maximal element at  $m - 1$  is swapped to its final destination  $m$ . Therefore, we have the following result.

```

PROCEDURE MergeForest(m)
  x := 1
  WHILE (2x + Reverse[x] < m)
    x := 2x + Reverse[x]
  IF (m-1 = x AND depth(x) = depth(m))
    x := x DIV 2
    WHILE (x > 0)
      Merge(m-1, x); x := x DIV 2
    Swap(m, m-1)
  ELSE
    WHILE (x > 0)
      Merge(m, x); x := x DIV 2

```

**Fig. 4.** The implementation of the Next-to- $m$  Strategy

**Theorem 2.** *Let  $k = \lceil \log n \rceil$ . The Next-to- $m$  strategy performs at most  $nk - 2^k + n - 2k$  comparisons.*

### 3.2 EXTERNAL-WEAK-HEAPSORT

Now we elaborate on the idea to bypass case a). In this section we assume an extension  $a[n \dots 2n - 1]$  to the element array. This violates the *In-Situ* condition for internal sorting and lifts the algorithm towards the the class of traditional *MERGESORT* variants that need additional element space of size  $O(n)$ . Moreover, other algorithms can also be tuned given this condition. For example to readjust the structure in *HEAPSORT* with respect to a new root element only one comparison between every two children on the special path is needed yielding a  $n \log n + O(n)$  algorithm. More precisely, from results of Doberkat (1984) and Knuth (1973) Cantone and Cincotti (2000) conclude that this so-called *EXTERNAL-HEAPSORT* algorithm performs at most  $n \log n + 2n$  comparisons in the worst case and at most  $n \log n + 1.88n$  comparisons on average.

In *EXTERNAL-WEAK-HEAPSORT* we further devise a Boolean vector **Active**, which is initialized with 1 and maintains the currently valid index positions. Since reverse bits are only required at inner nodes of the tree we may re-use the bits at the leaves to implement **Active**. If **Active**[ $i$ ] is 0, then the key at position  $i$  is swapped into the area  $a[n \dots 2n - 1]$ , and remains invalid for the current *Weak-Heap*. The positions in the output are controlled by a loop variable *call*. The algorithm and its *MergeForest* function are shown in Fig 5.

If the condition of case a) is fulfilled, the variable **temp** serves as the new root position and the special path is shortened by one. Since we have enough space the maximal element can be written to its final position and its position is deactivated. The main loop organizes the overall sorting process. Since we enforce case b) or c) for each *MergeForest* operation, we have the following result.

**Theorem 3.** *Let  $k = \lceil \log n \rceil$ . EXTERNAL-WEAK-HEAPSORT is an incrementally external sorting algorithm with  $nk - 2^k + 1$  comparisons in the best-, worst-, and average case. The number of transposition is bounded by  $nk - 2^k + 1$  and the number of assignments is  $n$ .*

```

PROCEDURE MergeForest(m)
  x := 1
  WHILE (2x + Reverse[x] < m)
    x := 2x + Reverse[x]
  IF (Active[x] = 0) x := x DIV 2
  IF (depth(x) = depth(m))
    temp := x; x := x DIV 2
  ELSE
    temp := m
  WHILE (x > 0)
    Merge(temp, x); x := x DIV 2
  a[2n - call] := a[temp]
  Active[temp] := 0

PROCEDURE EXTERNAL-WEAK-HEAPSORT
  WeakHeapify
  a[2n-1] := a[0]
  i := n-1
  call := 1
  WHILE (i > 1)
    WHILE (Active[i] = 0 AND i > 1)
      i := i - 1
    call := call + 1
    MergeForest(i)

```

Fig. 5. The implementation of the *EXTERNAL-WEAK-HEAPSORT* algorithm.

### 3.3 RELAXED-WEAK-HEAPSORT

The last section has shown that we loose information if we assign an element not on the special path to the root, and that it is better to shorten this path instead. To improve the space performance of the above algorithm is to relax the request of immediately placing the elements to the end of the array and to maintain a data structure in which these array positions are stored (Fig. 6): *WishToGo[i]* holds the index, where the element at  $i$  has to be stored and *WishToHave[i]* contains the index of the element, which has to be placed at position  $i$ , serving as a doubly connected linked list. With an additional Boolean flag for each element we indicate which one contains additional placement information. Therefore, we consume  $\log n$  bits only for those elements that cannot be placed onto their final position. Hence, the extra space requirement is proportional to the maximum amount of link information. For sake of brevity in the following implementation, we use a simple array representation.

```

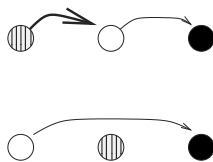
PROCEDURE MergeForest(m)
  x := 1
  WHILE (2x + Reverse[x] < m)
    x := 2x + Reverse[x]
  IF (Active[x] = 0) x := x DIV 2
  IF (depth(x) = depth(m))
    temp := x; x := x DIV 2
  ELSE
    temp := m
  WHILE (x > 0)
    Merge(temp, x); x := x DIV 2
  WishToHave[workidx] := temp
  WishToGo[temp] := workidx
  Active[temp] := 0

PROCEDURE RELAXED-WEAK-HEAPSORT
  WeakHeapify
  WishToHave[n-1] := 0
  WishToGo[0] := n-1
  workidx := n-2
  i := n-1
  WHILE (i > 1)
    WHILE (Active[i] = 0 AND i > 1)
      WishToHave[WishToGo[i]] := WishToHave[i]
      WishToGo[WishToHave[i]] := WishToGo[i]
      Swap(WishToHave[i], i)
      i := i - 1
    IF (i > 1)
      MergeForest(i)
      workidx := workidx - 1
  Swap(0,1)

```

Fig. 6. The implementation of the *RELAXED-WEAK-HEAPSORT* algorithm.

If in the main loop the bit *Active[i]* becomes 0, we swap the elements at *WishToHave[i]* and  $i$  and adjust the pointers accordingly (Fig. 7). When the entire level has passed its sorting phase, all lasting requests are executed. Moreover, in these situations, all space for the links will be released.



**Fig. 7.** Modification of pointers, while executing a data movement. Each node represents an array element. An edge from one node to another is drawn, if the latter is located at the final position of the former. The bold edge indicates the exchange to be fulfilled due to both elements being inactive.

**Theorem 4.** Let  $k = \lceil \log n \rceil$ . *RELAXED-WEAK-HEAPSORT* consumes at most  $O(n \log n)$  extra bits and executes exactly  $nk - 2^k + 1$  comparisons in the best-, worst-, and average case. The number of transposition is bounded by  $(k + 1)n - 2^k + 1$ . There is no data assignment.

Since scalable worst-case examples of Edelkamp and Wegener always enforce case a) or b), for each level no short path will be encountered. After deleting a few root elements the distribution of elements in a heap structure is hard to predict (Doberkat 1984). Hence, theoretical considerations on the average space consumption are difficult to obtain. In the experiments we will see that at most  $O(n)$  bits or  $O(n/\log n)$  link indices suffice on the average.

**Conjecture** *RELAXED-WEAK-HEAPSORT* consumes at most  $O(n)$  bits on the average.

We have some backing arguments for the conjecture. Since the average case of *WEAK-HEAPSORT* has been estimated at  $n \log n + d(n)n$  comparisons with  $d(n) \in [-0.47, -0.42]$ , the expected number of comparison is close to the average of the worst case ( $n \log n + 0.1n$ ), and the best case ( $n \log n - 0.9n$ ). Therefore, we might assume that in the *MergeForest* operation  $m$  is as likely to be in the same depth than  $x$  as not to be. Therefore, the probability of  $m$  becoming inactive in one Merge-Forest step is about 1/2 and bottlenecks for aimed transpositions at the end of the array are likely to be eliminated early in the sorting process.

### 3.4 GREEDY-WEAK-HEAPSORT

A further improvement to the extra space needed in *RELAXED-WEAK-HEAPSORT* is to fulfill any desire as soon as possible. Instead of waiting for the active element with the highest index to become inactive, we frequently rearrange the link structure as indicated in the pseudo code of Fig. 8. The procedure `collapse(i)` checks if some element can be placed at position  $i$ , if the final position of the element at  $i$  is free, or both, and adjusts the pointers accordingly. As shown in Fig. 9 a circular shift may save some pointer assignments. Since `collapse` shrinks the doubly connected list structure, *GREEDY-WEAK-HEAPSORT* consumes less additional data links than *RELAXED-WEAK-HEAPSORT*.

```

PROCEDURE MergeForest(m)
  x := 1
  WHILE (2x + Reverse[x] < m)
    x := 2x + Reverse[x]
  IF (Active[x] = 0) x := x DIV 2
  IF (depth(x) = depth(m))
    temp := x; x := x DIV 2
  ELSE
    temp := m
  WHILE (x > 0)
    Merge(temp, x); x := x DIV 2
  WishToHave[workidx] := temp
  WishToGo[temp] := workidx
  Active[temp] := 0
  collapse(temp)

PROCEDURE GREEDY-WEAK-HEAPSORT
  WeakHeapify
  WishToHave[n-1] := 0
  WishToGo[0] := n-1
  workidx := n-2
  i := n-1
  WHILE (i > 1)
    WHILE (Active[i] = 0 AND i > 1)
      collapse(i)
      i := i - 1
    IF (i > 1)
      MergeForest(i);
      workidx := workidx - 1
  Swap(0,1)

```

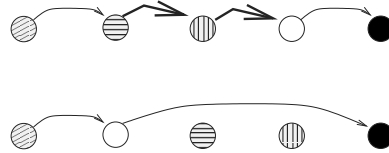
Fig. 8. The implementation of *GREEDY-WEAK-HEAPSORT*.

Fig. 9. Modification of pointers, while fulfilling two desires at a time. The bold edges indicate transpositions to be executed.

#### 4 QUICK-WEAK-HEAPSORT

With *QUICK-HEAPSORT* Cantone and Cincotti (2000) present an efficient hybrid sorting algorithm based on combining the Divide-and-Conquer idea of partitioning the element array of *QUICKSORT* with *HEAPSORT*. The trick is to split the array according to the pivot  $P$  in a reverse way. More precisely, the array  $a[1..n]$  is partitioned into two sub-arrays:  $a[1..p-1]$  and  $a[p+1..n]$ . In these sub-arrays,  $a[p] = P$  and the keys in  $a[1..p-1]$  must be larger than or equal to  $P$ , and the keys in  $a[p+1..n]$  must be smaller than or equal to  $P$ .

Now *EXTERNAL-HEAPSORT* is invoked on the sub-array of smaller cardinality. If this is the first part of the array, *QUICK-HEAPSORT* builds a max-heap, otherwise it constructs a min-heap. We consider the first case only, since the latter case is dealt analogously. The construction of the heap is the same as in traditional *HEAPSORT*. If in the sorting phase a root element  $r$  has been extracted it is repeatedly replaced with the larger child until we reach a leaf position. Then this position can safely be refilled with the element located at the final position of  $r$ , since it will be smaller than or equal to the pivot. If the sorting phase of *EXTERNAL-HEAPSORT* terminates, the refined partitioning phase results in three sub-arrays: The elements in the first two parts are smaller than or equal to the pivot, and the right-most region will contain the elements formerly found at  $a[1..p-1]$  in ascending order. Hence, the pivot is moved to its correct place and the left side of the array is recursively sorted.

Cantone and Cincotti derived a worst case bound of  $n \log n + 3n$  comparisons on the average. *QUICK-WEAK-HEAPSORT* is an elaboration on *QUICK-HEAP-*

*SORT*. The algorithm is obtained by exchanging the *EXTERNAL-HEAPSORT* component by *EXTERNAL-WEAK-HEAPSORT*. Since the (special) path determined by the smaller child elements in a heap of size  $n$  is of length  $\lfloor \log n \rfloor$  or  $\lfloor \log n \rfloor - 1$  the major improvement from turning to the *WEAK-HEAPSORT* variant is the optimal construction time in  $n - 1$  comparisons. In the following, we mimic the analysis of Cantone and Cincotti to achieve the stated result of at most  $n \log n + 0.2n + o(n)$  comparisons in the average case.

**Lemma 1.** (*Cantone and Cincotti*) Let  $C^*(n) = n \log n + \alpha n$  and  $f_1(n), f_2(n)$  be functions of type  $\beta n + o(n)$  for all  $n \in \mathbb{N}$  and  $\alpha, \beta \in \mathbb{R}$ . The solution to the following recurrence equations, with initial conditions  $C(1) = 0$  and  $C(2) = 1$ :

$$C(2n) = \frac{1}{2n} [(2n+1)C(2n-1) - C(n-1) + C^*(n-1) + f_1(n)], \quad (1)$$

$$C(2n+1) = \frac{1}{2n+1} [(2n+2)C(2n) - C(n) + C^*(n) + f_2(n)] \quad (2)$$

for all  $n \in \mathbb{N}$ , is  $C(n) = n \log n + (\alpha + \beta - 2.8854)n + o(n)$ .

For *QUICK-WEAK-HEAPSORT* we have  $\alpha = -0.9139$  and  $\beta = 4$ . Hence, the average case number of comparisons is bounded by  $n \log n + 0.2n + o(n)$  as stated in the following theorem.

**Theorem 5.** On the average *QUICK-WEAK-HEAPSORT* sorts  $n$  elements in at most  $n \log n + 0.2n + o(n)$  comparisons.

*CLEVER-HEAPSORT* and *CLEVER-WEAK-HEAPSORT* are obtained by applying a median-of-three strategy in the partition phase.

## 5 Experiments

For small values of  $n$ ,  $2 \leq n \leq 11$ , we have compared the number of comparisons in *EXTERNAL-WEAK-HEAPSORT* with the lower bound. For  $n \in \{2, 3, 4\}$  the number of comparisons matches the lower bound of  $\lceil \log n! \rceil$  comparisons. For larger values of  $n \in \{5, 6, 7, 8\}$  we are off by one, and for  $n \in \{9, 10, 11\}$  we are off by exactly only two comparisons.

Given a uniform distribution of all permutations of the input array, *QUICK-SORT* (Hoare (1962)) reveals an average number of at most  $1.386n \log n - 2.846n + O(\log n)$  comparisons. In *CLEVER-QUICKSORT*, the median-of-three variant of *QUICKSORT*, this value is bounded by  $1.188n \log n - 2.255n + O(\log n)$ .

*BOTTOM-UP-HEAPSORT* (Wegener (1993)) is a variant of *HEAPSORT* with  $1.5n \log n + \Theta(n)$  key comparisons in the worst case. The idea is to search the path to the leaf independently to the place for the root element to sink. Since the expected depth is high, this path is traversed bottom-up. The average number of comparisons in *BOTTOM-UP-HEAPSORT* can be bounded by  $n \log n + O(n)$ . A further refinement *MDR-HEAPSORT* proposed by McDiarmid and Reed (1989), performs less than  $n \log n + 1.1n$  comparisons in the worst case (Wegener (1993)).



**Table 1.** Number of comparisons, exchanges, assignments of the different algorithms on random data averaged over 30 runs.

Comparisons	10	100	1000	10000	100000	1000000
<i>QUICKSORT</i>	30.56	729.10	11835.10	161168.63	2078845.53	25632231.16
<i>CLEVER-QUICKSORT</i>	28.86	649.36	10335.83	143148.86	1826008.56	22290868.50
<i>BOTTOM-UP-HEAPSORT</i>	32.43	694.76	10305.10	136513.60	1698272.06	20281364.60
<i>MDR-HEAPSORT</i>	30.76	656.43	9886.83	132316.06	1656365.03	19863064.60
<i>WEAK-HEAPSORT</i>	27.60	618.63	9513.40	128568.30	1618689.90	19487763.03
<i>RELAXED-WEAK-HEAPSORT</i>	25.00	573.00	8977.00	123617.00	1568929.00	18951425.00
<i>GREEDY-WEAK-HEAPSORT</i>	25.00	573.00	8977.00	123617.00	1568929.00	18951425.00
<i>QUICK-HEAPSORT</i>	29.66	781.83	11630.00	150083.96	1827962.60	22000593.93
<i>QUICK-WEAK-HEAPSORT</i>	28.43	660.90	10045.76	133404.13	1659777.23	20266957.30
<i>CLEVER-HEAPSORT</i>	29.70	727.73	11322.86	148107.30	1819264.86	21416210.80
<i>CLEVER-WEAK-HEAPSORT</i>	27.86	609.23	9418.96	128164.83	1614333.26	19602152.26

Exchanges	10	100	1000	10000	100000	1000000
<i>QUICKSORT</i>	9.36	161.00	2369.70	31501.43	391003.10	4668411.06
<i>CLEVER-QUICKSORT</i>	8.53	165.33	2434.03	32294.80	401730.13	4803056.93
<i>BOTTOM-UP-HEAPSORT</i>	9.00	99.00	999.00	9999.00	99999.00	999999.00
<i>MDR-HEAPSORT</i>	9.00	99.00	999.00	9999.00	99999.00	999999.00
<i>WEAK-HEAPSORT</i>	15.50	336.66	4969.70	65037.53	803705.33	9578990.20
<i>RELAXED-WEAK-HEAPSORT</i>	22.76	412.06	5800.13	75001.40	915744.06	10790844.36
<i>GREEDY-WEAK-HEAPSORT</i>	20.00	357.30	5187.33	69255.83	856861.66	10177597.60
<i>QUICK-HEAPSORT</i>	7.63	72.20	670.80	6727.60	65321.76	696398.30
<i>QUICK-WEAK-HEAPSORT</i>	13.46	313.00	4990.66	66929.90	838173.16	9985011.30
<i>CLEVER-HEAPSORT</i>	7.46	70.70	661.00	6478.16	64919.86	652938.26
<i>CLEVER-WEAK-HEAPSORT</i>	14.53	326.66	5071.36	67578.40	842446.46	10066407.13

Assignments	10	100	1000	10000	100000	1000000
<i>QUICKSORT</i>	6.33	66.00	666.70	6670.86	66651.43	666649.50
<i>CLEVER-QUICKSORT</i>	4.33	46.26	462.20	4571.30	45745.93	457071.86
<i>BOTTOM-UP-HEAPSORT</i>	43.86	781.66	11116.30	144606.73	1779038.73	21088698.53
<i>MDR-HEAPSORT</i>	43.50	769.00	10965.56	142759.63	1758266.06	20856586.16
<i>WEAK-HEAPSORT</i>	11.00	101.00	1001.00	10001.00	100001.00	1000001.00
<i>RELAXED-WEAK-HEAPSORT</i>	0.00	0.00	0.00	0.00	0.00	0.00
<i>GREEDY-WEAK-HEAPSORT</i>	5.20	126.80	1288.40	11649.73	118028.26	1226882.80
<i>QUICK-HEAPSORT</i>	32.93	693.70	10752.93	141534.50	1757454.76	20697840.96
<i>QUICK-WEAK-HEAPSORT</i>	4.46	11.46	18.90	25.30	33.26	42.76
<i>CLEVER-HEAPSORT</i>	30.16	716.70	10827.53	142134.60	1759835.10	20892687.56
<i>CLEVER-WEAK-HEAPSORT</i>	2.96	8.50	14.36	20.70	26.16	32.93

by using one bit to encode on which branch the smaller element can be found and another one to mark if this information is unknown.

Table 1 presents data of our `c++`-implementation of various performant sorting algorithms on some random sets of floating-point data. We have referenced the number of data comparisons, transpositions and assignments.

If  $f^0(x) = x$  and  $f^n(x) = \ln(f^{n-1}(x+1))$  then Table 2 depicts the running times in sorting one million floating point elements on a Pentium III/450Mhz (Linux, `g++ -O2`) according to modified comparisons with  $f^n$  applied to both arguments,  $0 \leq n \leq 7$ .

The variance in performance for *QUICKSORT* variants is large while in *HEAPSORT* variants we observed a very small value.

The remaining Table 3 backs the given conjecture on the  $O(n/\log n)$  bound for the number of lasting transpositions in *RELAXED-WEAK-HEAPSORT* and *GREEDY-WEAK-HEAPSORT*.

**Table 2.** Time performance according to different comparison functions in sorting one million floating point elements.

	$f^0$	$f^1$	$f^2$	$f^3$	$f^4$	$f^5$	$f^6$	$f^7$
<i>QUICKSORT</i>	3.86	14.59	26.73	39.04	51.47	63.44	75.68	87.89
<i>CLEVER-QUICKSORT</i>	3.56	12.84	23.67	33.16	43.80	54.37	64.62	75.08
<i>BOTTOM-UP-HEAPSORT</i>	5.73	13.49	22.60	32.05	41.59	51.14	60.62	70.11
<i>MDR-HEAPSORT</i>	7.14	15.39	24.37	33.82	43.04	52.63	61.87	71.02
<i>WEAK-HEAPSORT</i>	7.15	14.89	23.66	32.82	41.97	51.08	60.13	69.27
<i>RELAXED-WEAK-HEAPSORT</i>	8.29	15.96	24.63	33.51	42.32	51.33	60.06	68.83
<i>GREEDY-WEAK-HEAPSORT</i>	9.09	16.60	25.24	34.12	43.03	51.77	60.72	69.78
<i>QUICK-HEAPSORT</i>	6.35	15.89	26.20	36.98	47.59	58.16	69.37	79.59
<i>QUICK-WEAK-HEAPSORT</i>	6.06	14.49	23.86	33.49	43.30	52.99	62.85	72.54
<i>CLEVER-HEAPSORT</i>	5.30	14.01	23.66	33.65	43.95	53.79	63.60	73.94
<i>CLEVER-WEAK-HEAPSORT</i>	5.97	13.82	22.83	31.95	41.31	50.40	59.58	69.61

**Table 3.** Worst case number of additional data links in 30 trials with respect to different numbers of data elements.

	10	100	1000	10000	100000	1000000
<i>RELAXED-WEAK-HEAPSORT</i>	6.00	28.00	229.00	1933.00	14578.00	44061.00
<i>GREEDY-WEAK-HEAPSORT</i>	2.00	11.00	124.00	1022.00	8533.00	28012.00
$n/\log n$	3.01	15.05	100.34	752.57	6020.60	50171.67

## 5.1 Conclusion

We have improved the *WEAK-HEAPSORT* algorithm by achieving the best case in every *MergeForest* step. *RELAXED-WEAK-HEAPSORT* requires  $n\lceil\log n\rceil - 2^{\lceil\log n\rceil} + 1 \leq n\log n - 0.9n$  comparisons which is only about  $0.5n$  more than necessary. Note that the number of comparisons  $(n-1) + \sum_{i=2}^{n-1} (\lceil\log(i+1)\rceil - 1) = \sum_{i=1}^{n-1} \lceil\log(i+1)\rceil$  corresponds exactly to the performance of traditional *INSERTIONSORT* (Steinhaus (1958)). Further on, in case of  $n = 2^k$  this number reduces to  $n\log n - n + 1$ . Recall that *WEAK-HEAPSORT* itself has a worst case number of about  $n\log n + 0.1n$  comparisons in the worst case, i.e., we save about  $n$  comparisons. We conjecture that *RELAXED-WEAK-HEAPSORT* can be implemented comprising  $O(n)$  bits on the average. If this space is not available we contribute a new hybrid *QUICKSORT*-related algorithm called *QUICK-WEAK-HEAPSORT* that performs at most  $n\log n + 0.2n + o(n)$  comparisons on the average. The algorithm uses *EXTERNAL-WEAK-HEAPSORT* as a subroutine and compares well with other known improvements to *QUICKSORT*. Van Emde (1970) has shown that no median strategy can lead to  $n\log n + o(n)$  comparisons on the average. Currently, *CLEVER-WEAK-HEAPSORT* can be judged to be the fastest *QUICKSORT* variant and *RELAXED-WEAK-HEAPSORT* to be the fastest *HEAPSORT* variant. As a challenge, Reinhardt (1992) shows that *MERGESORT* can be designed *In-Situ* with  $n\log n - 1.3n + O(\log n)$  comparisons in the worst case. However, for practical purposes these algorithms are too slow. The number of comparisons, assignments and exchanges as analyzed in this paper are good indicators for the running time of sorting algorithms. Concerning the layered memory and cache structure within a modern personal

computer, a better prediction requires a so-called *meticulous* analysis, which in turn, can lead to more efficient algorithms. A good example for this technique is the heap construction phase, analysed and improved by Bojesen et al. (1999).

## Acknowledgements

We thank Jessica Harrington for proofreading and the anonymous referees for helpful comments.

## References

1. J. Bojesen, J. Katajainen, and M. Spork. Performance engineering case study: Heap construction. *WAE, LNCS*, pages 301–315, 1999.
2. D. Cantone and G. Cinotti. QuickHeapsort, an efficient mix of classical sorting algorithms. *CIAC, LNCS*, 1767:150–162, 2000.
3. E. E. Doberkat. An average case analysis of Floyd’s algorithm to construct heaps. *Information and Control*, 61(2):114–131, 1984.
4. R. D. Dutton. The weak-heap data structure. Technical report, University of Central Florida, Orlando, FL 32816, 1992.
5. R. D. Dutton. Weak-heap sort. *BIT*, 33:372–381, 1993.
6. S. Edelkamp and I. Wegener. On the performance of *WEAK-HEAPSORT*. *STACS, LNCS*, pages 254–266, 2000.
7. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithm. *Journal of the ACM*, 34(3):596–615, 1987.
8. C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.
9. D. Knuth. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.
10. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Text and Monographs in Computer Science. Springer-Verlag, 1993.
11. C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *Journal of Algorithms*, 10:352–365, 1989.
12. S. Nilsson. *Radix Sorting & Searching*. PhD thesis, Lund University, 1996.
13. A. Papernov and G. Stasevich. The worst case in shellsort and related algorithms. *Problems Inform. Transmission*, 1(3):63–75, 1965.
14. K. Reinhardt. Sorting in-place with a worst case complexity of  $n \log n - 1.3n + O(\log n)$  comparisons and  $en \log n + O(1)$  transports. *LNCS*, 650:489–499, 1992.
15. H. Steinhaus. *One hundred problems in elementary mathematics (Problems 52,85)*. Pergamon Press, London, 1958.
16. M. H. van Emden. Increasing the efficiency of QUICKSORT. *Communications of the ACM*, 13:563–567, 1970.
17. I. Wegener. The worst case complexity of McDiarmid and Reed’s variant of BOTTOM-UP HEAPSORT is less than  $n \log n + 1.1n$ . *Information and Computation*, 97(1):86–96, 1992.
18. I. Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if  $n$  is not very small). *TCS*, 118:81–98, 1993.

# Efficient Sorting Using Registers and Caches

Lars Arge, Jeff Chase, Jeffrey S. Vitter, and Rajiv Wickremesinghe\*

Department of Computer Science, Duke University, Durham, NC 27708, USA  
`rajiv@cs.duke.edu`

**Abstract.** Modern computer systems have increasingly complex memory systems. Common machine models for algorithm analysis do not reflect many of the features of these systems, e.g., large register sets, lockup-free caches, cache hierarchies, associativity, cache line fetching, and streaming behavior. Inadequate models lead to poor algorithmic choices and an incomplete understanding of algorithm behavior on real machines.

A key step toward developing better models is to quantify the performance effects of features not reflected in the models. This paper explores the effect of memory system features on sorting performance. We introduce a new cache-conscious sorting algorithm, R-MERGE, which achieves better performance in practice over algorithms that are theoretically superior under the models. R-MERGE is designed to minimize memory stall cycles rather than cache misses, considering features common to many system designs.

## 1 Introduction

Algorithm designers and programmers have traditionally used the *random access machine* (RAM) model to guide expectations of algorithm performance. The RAM model assumes a single processor with an infinite memory having constant access cost. It is well-understood that the RAM model does not reflect the complexity of modern machines, which have a hierarchy of successively slower and larger memories managed to approximate the illusion of a large, fast memory. The levels of the memory hierarchy often have varying characteristics (eg: fully-associative vs. direct-mapped, random vs. sequential/blocked access).

Program performance is largely determined by its interaction with the memory system. It is therefore important to design algorithms with access behavior that is efficient on modern memory systems. Unfortunately, the RAM model and its alternatives are inadequate to guide algorithmic and implementation choices on modern machines. In recent years there has been a great deal of progress in devising new models that more accurately reflect memory system behavior. These models are largely inspired by the theory of *I/O Efficient Algorithms* (also known as *External Memory Algorithms*), based on the I/O model of Aggarwal and Vitter [2]. The I/O model has yielded significant improvements for

---

\* This work supported in part by the National Science Foundation through ESS grant EIA-9870734 and by the U.S. Army Research Office through MURI grant DAAH04-96-1-0013.

I/O-bound problems where the access gaps between levels of the hierarchy (e.g., memory and disk) are largest.

The difference in speed between upper levels of the memory hierarchy (CPU, caches, main memory) continues to widen as advances in VLSI technology widen the gap between processor and memory cycle time. It is not the number of steps or instructions, but the number of cache misses, and the duration of the resulting stalls that determine the execution time of a program. This contrasts with the models derived from the I/O model, which only count the number of cache misses (analogous to number of I/Os in the I/O model).

This makes it increasingly important for programs to make efficient use of caches and registers. It is therefore important to extend the benefits of the I/O model to incorporate the behavior of caches and registers. However, cache behavior is difficult to model adequately. Unlike main memory, caches are faster and tend to have low associativity. The former makes computationally expensive optimizations ineffective, and the latter increases the difficulty of modeling the memory. The complexity of memory systems, including cache hierarchies, prefetching, multiple- and delayed- issue processors, lockup-free caches, write buffers and TLBs (see Section 2) make it difficult to analyze different approaches theoretically. Thus empirical data is needed to guide the development of new models.

This paper makes three contributions. First, we outline key factors affecting memory system performance, and explore how they affect algorithmic and implementation choices for a specific problem: sorting. Second, we present quantitative data from several memory-conscious sort implementations on Alpha 21x64 CPUs in order to quantify the effects of these choices. These results also apply in general to other architectures including Intel Pentium and HP PA-RISC (see appendix). Finally, we show how a broader view of memory system behavior leads to a sorting implementation that is more efficient than previous approaches on the Alpha CPU, which is representative of modern RISC processors. This sorting approach, R-MERGE, benefits from optimizations that would not be considered under a simple model of cache behavior, showing that a straightforward extension of the I/O model is inadequate for memory systems. In particular, R-MERGE uses a hybrid mergesort/quicksort approach that considers the duration of stalls, streaming behaviour, etc., rather than merely minimizing cache misses. R-MERGE also shows the importance of efficient register usage during the merge phase. We generalize from our experience to propose key principles for designing and implementing memory-conscious algorithms.

This paper is organized as follows. Section 2 briefly reviews memory system structure and extracts principles for memory-conscious algorithm design and implementation. Section 3 and Section 4 apply these principles to the problem of sorting, describe the memory-conscious sort implementations used in our experiments, and present experimental results from these programs. Section 5 sets our work in context with related work. We conclude in Section 6.

## 2 Models of Memory Systems

Modern memory systems typically include several levels of memory in a hierarchy. The processor is at level 0, and has a small number of *registers* that store

individual data items for access in a single cycle. One contribution of this paper is to show the benefits of incorporating explicit register usage into algorithm design. Register usage is usually managed by the compiler. Careful use of registers can reduce memory accesses and instructions executed by reducing the number of load/store instructions required.

*Caches* are small, fast, stores of “useful” data. In today’s systems, caches are managed by the memory system rather than the program. Several levels of cache are placed between main memory and the processor. Elements are brought into the cache in *blocks* of several words to make use of spatial locality. There are many hardware devices that help improve performance, but they also complicate analysis. A *stall* occurs when the processor is forced to wait for the memory system instead of executing other (independent) instructions. A *Write buffer* reduces the number of write-stalls the processor suffers. It allows the processor to continue immediately after the data is written to the buffer, avoiding the miss penalty. *Translation Look-aside Buffers* (TLBs) provide a small cache of address translations in a hardware lookup table. Accessing large numbers of data streams can incur performance penalties because of TLB thrashing [12].

The I/O model of Aggarwal and Vitter [2] is important because it bridges the largest gap in the memory hierarchy, that between main memory and disk. The model describes a system with a large, slow memory (disks) and a limited fast memory (main memory). The fast memory is fully-associative and data is transferred to and from it in blocks of fixed size (an *I/O operation*). Computation is performed on data that is in fast memory. Computation time is usually not taken into account because the I/O time dominates the total time. Algorithms are analyzed in this model in terms of number of I/O operations.

Others [11,12,16] have proposed cache models similar to the I/O model. The cache is analogous to the (fast) main memory in the I/O model. Main memory takes the place of disks, and is assumed to contain all the data. The following parameters are defined in the model:

- N = number of elements in the problem instance
- B = number of elements in a cache block
- C = number of blocks in the cache

The capacity of the cache is thus  $M = BC$  elements. A block transfer takes place when a cache miss occurs. The I/O model is not directly applicable to caches because of several important differences, described below.

- Computation and main memory access cost can be ignored in the I/O model because the I/O cost dominates. The access times of caches and main memory are much closer, so the time spent processing becomes significant.
- Modern processors can have multiple outstanding memory operations, and may execute instructions out of order. This means that a cache miss does not increase the execution time provided other instructions are able to execute while the miss is handled. Memory accesses overlapped with computation do not contribute toward total execution time.
- Modern memory systems are optimized for sequential accesses. An algorithm that performs sequential (streaming) accesses has an advantage over another that accesses the blocks randomly, even if both have the same total number of accesses or cache misses.

- The cache (in our case) is direct-mapped, whereas the main memory is fully-associative. This restricts how we use the cache, since the cache location used is determined by the location in memory where the data element is stored. There are no special instructions to control placement of data in the cache.
- The model only considers one cache; in practice there is a hierarchy of caches.
- Registers can be used to store the most frequently accessed data. They form what can be viewed as a very small, fast, controllable cache. Algorithmic and programming changes are usually required to exploit registers.
- The size of the TLB limits the number of concurrent data streams that can be handled efficiently [12].

Using cache misses as a metric can give misleading results, because only the time spent *waiting* for the memory system (memory stalls) contributes to the total time. The number of cache misses alone does not determine the stall time of a program. A program that spends many cycles computing on each datum may have very few memory stalls, whereas a program that just touches data may be dominated by stalls. We are interested in the total running time of the program which is dependent on *both* instructions executed and duration of memory system *stalls* (not number of misses). A more realistic measure of CPU time takes into account the memory stall time and the actual execution time [7]:

$$\text{CPU time} = (\text{CPU execution cycles} + \text{memory stall cycles}) \times \text{Clock cycle time}$$

To consider the effects of these factors, we experiment with hybrid sorting algorithms. Our designs are guided by augmenting the cache model with three principles.

1. The algorithm should be instruction-conscious *and* cache-conscious: we need to balance the conflicting demands of reducing the number of instructions executed, and reducing the number of memory stalls.
2. Memory is not uniform; carefully consider the access pattern. Sequential accesses are faster than random accesses.
3. Effective use of registers reduces instruction counts as well as memory accesses.

### 3 Sorting

We consider the common problem of sorting  $N$  words (64-bit integers in our case) for values of  $N$  up to the maximum that will fit in main memory. This leads us to make several design choices.

1. We observe that quicksort does the least work per key as it has a tight inner loop, even though it is not cache-efficient. It also operates in place, allowing us to make full use of the cache. Therefore it is advantageous to sort with quicksort once the data is brought into the cache. Our mergesorts combine the advantage of quicksort with the cache efficiency of merging by forming cache-load-sized runs with quicksort. For example, when we compared two versions of our R-MERGE algorithm—one that does one pass of order  $k = N/(BC)$ , and one that does two passes of order  $k = \sqrt{N/(BC)}$  but

consequently has more efficient code—the two-pass version had double the misses in the merge phase, but executed 22% fewer instructions and was 20% faster. The run formation was identical for both versions. This indicates that it is necessary to focus on instruction counts beyond a certain level of cache optimization.

2. The streaming behavior of memory systems favours algorithms that access the data sequentially, e.g., merging and distribution (multi-way partitioning). Merging has an overall computational advantage over distribution because layout is not dependent on the ordering of the input data. This eliminates the extra pass needed by distribution methods to produce sequential output, and more than offsets the extra work required to maintain a dynamic merge heap, compared to a fixed search tree.
3. The third key to good performance is the efficient use of registers. The merge heap is an obvious candidate, as it is accessed multiple times for every key, and is relatively small. Storing the heap in registers reduces the number of instructions needed to access it, and also avoids cache interference misses, though this effect is small at small order [16], by eliminating memory accesses in the heap. However, the small number of registers limits the order ( $k$ ) of the merge. The small order merge executes less instructions, but also causes more cache misses. Because the processor is not stalling while accessing memory, the increase in misses does not lead to an increase in memory stalls, and so the overall time is decreased. The number of instructions is proportional to  $N \times (\text{average cost of heap operation}) \times \log_k N$ . The average cost of a heap operation averaged over the  $\log k$  levels of the heap is less if the heap is smaller. The number of misses is  $(N/B) \lceil \log_k(N/BC) \rceil$ .

These factors lead us to use a hybrid mergesort/quicksort algorithm. We optimized the core merge routine at the source level to minimize the number of memory locations accessed, and especially to reduce the number of instructions executed. Runs shorter than cache size are used, and the input is padded so that all runs are of equal same size, and so that the full order of runs are merged. This appears to be the most efficient arrangement. We found merging to be at least 10% faster than the corresponding distribution sort.

Our hybrid algorithm is as follows. The merge order,  $k$ , is input into the program.

1. Choose number of passes,  $\lceil \log_k(N/BC) \rceil$ .
2. Form initial runs (which are approximately of length  $BC$  elements) using quicksort (in cache).
3. Merge sets of  $k$  runs together to form new runs.
4. Repeat merge step on new runs until all elements are in one run.

The merge itself is performed using techniques based on the selection tree algorithms from [8].

1. Take the first element from each of the  $k$  runs and make a heap. Each element of the heap consists of the value, `val`, and pointer to the next element of the run, `nextptr`.
2. Insert a sentinel (with key value  $+\infty$ ) after the last element of each run. The sentinel eliminates end of run checks in the inner loop. Note that the previous step created holes in the data at just the right places.



3. Repeat  $N$  times: output the min; obtain the next item in same run and replace the min; heapify. We halve the number of calls to heapify by not maintaining the heap property between the 'extract min' and insert.

```

*output++ ← min.val
min.val ← *min.nextptr++
heapify()

```

We refer to this algorithm as Q-MERGE. One advantage of Q-MERGE is that with low merge orders it is possible to store the merge heap in registers, further improving performance. The next section shows that this approach—R-MERGE—is up to 36% faster than previous comparison-sorting algorithms.

It is essential to engineer the inner loop(s) so that the compiler can generate efficient code. Duplicating compiler optimizations (eg: loop unrolling, instruction reordering, eliminating temporary variables) is ineffective, and could actually hurt overall performance. This is partly because the compiler has more detailed knowledge of the underlying machine. Although merging is not done in-place, cache performance is not degraded because memory writes are queued in the write buffer, and do not incur additional penalties.

Data and control dependencies slow program execution by limiting the degree of parallelism (modern processors are superscalar—capable of executing multiple instructions in parallel). We structure the code to reduce dependencies as much as possible (for example, by avoiding extraneous checking within the inner loop). However, comparison based sorting is inherently limited by the necessity of performing  $\Theta(n \log n)$  *sequential* comparisons. (Successful branch prediction and other techniques could reduce the actual number of sequential comparisons required).

## 4 Experiments

We performed most of our experiments on a Digital Personal Workstation 500au with a 500MHz Alpha 21164 CPU. We also validated our results on several other architectures (see appendix): (1) Alpha 21264 (500MHz) with a 4MB cache. (2) Alpha 21264A (467MHz) with a 2MB cache. (3) Intel Pentium III (500MHz) with a 512K cache. The 21164 is a four-way superscalar processor with two integer pipelines. The memory subsystem consists of a two-level on-chip data cache and an off-chip L3 cache. It has 40 registers (31 usable). The memory subsystem does not block on cache misses, and has a 6-entry  $\times$  32-byte block write buffer that can merge at the peak store issue rate. The L1 cache is 8K direct-mapped, write-through. The L2 cache is 96KB, 3-way associative, write-back. The L3 cache (which is the only one we directly consider in this paper) is 2MB ( $C = 256K$ ), direct-mapped, write-back, with 64-byte blocks ( $B = 8$ ). The access time for the off-chip cache is four CPU cycles. Machine words are 64-bits (8-bytes) wide. The TLB has 64 entries [6].

We performed cache simulations and measured the actual number of executed instructions by instrumenting the executables with the ATOM [5] tool. We used the hardware counters of the Alpha 21164 and the `dcpi` tool [3] to count memory stalls. This tool was also invaluable in optimizing the code as it allows instruction level tracking of stalls. All the programs were written in C, and compiled with the vendor compiler with optimizations enabled.

We compare our programs with the original cache-conscious programs of LaMarca and Ladner [11] described below.

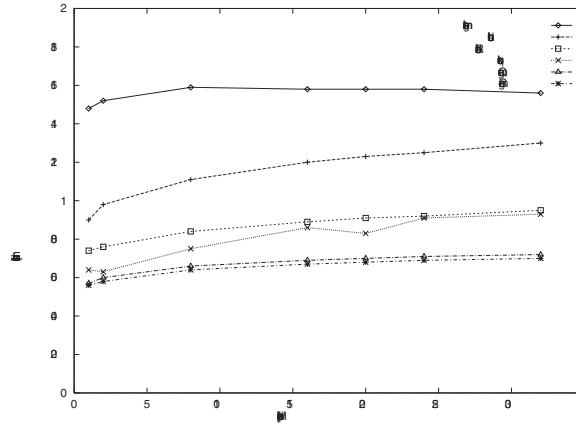
- TILED MERGESORT is a tiled version of an iterative binary mergesort. Cache-sized tiles are sorted using mergesort, and then merged using an iterative binary merge routine. Tiling is only effective over the initial  $\Theta(\log(BC))$  passes.
- MULTI-MERGESORT uses iterative binary mergesort to form cache-sized runs and then a single  $\Theta(N/BC)$  way merge to complete the sort.
- QUICKSORT is a memory-tuned version of the traditional optimized quicksort. It has excellent cache performance within each run, and is most efficient in instructions executed (among comparison-based sorts). However, quicksort makes  $\Theta(\log N)$  passes, and thus incurs many cache misses. Memory-tuned quicksort sorts small subsets (using insertion sort) when they are first encountered (and are already in the cache). This increases the instruction count, but reduces the cache misses. To make our analysis more conservative, we improved this version by simplifying the code to rely more on compiler optimization (eg: *removing* programmer-unrolled loops), yielding better performance.
- DISTRIBUTION SORT uses a single large multi-way partition pass to create subsets which are likely to be cache-sized. Unlike quicksort, this is not in-place, and also has a higher instruction overhead. However, cache performance is much improved. This is similar to the Flashsort variant of Rahman and Raman [12].

Our programs are as follows.

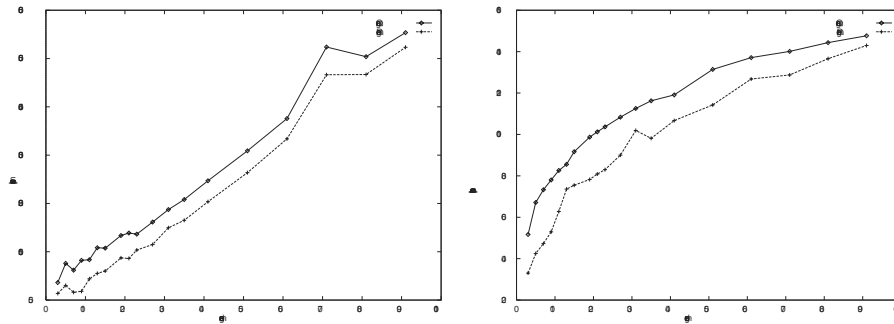
- R-DISTRIBUTION is a register-based version of distribution sort with a counting pass. It limits the order of the distribute in order to fit the pivots and counters in registers. This leads to an increased number of passes over the data.
- R-MERGE is one of two programs that implement the hybrid mergesort/quicksort algorithm described in Section 3: R-MERGE uses small merge order and direct addressing (discussed below), allowing the compiler to assign registers for key data items (the merge heap in particular).
- Q-MERGE also implements the hybrid algorithm. However, it uses an array to store the merge heap. We use this more traditional version to determine the effect of using registers.

Both the merge programs have heap routines custom-written for specific merge orders. These routines are generated automatically by another program. We also evaluated a traditional heap implementation.

It is possible to access a memory location by *direct addressing* or by *indirect addressing*. In the former, the address of the operand is specified in the instruction. In the latter, the address has to be computed, for example as an offset to a base; this is frequently used to access arrays. In a RISC processor like the Alpha 21164, indirect accesses to memory usually take two instructions: one to compute the address, and one to access the data. R-MERGE tries to be register efficient and thus uses direct addressing and a small merge order  $k$  so that the



**Fig. 1.** Comparison with cache-conscious sort programs of LaMarca and Ladner [11]. Time per key ( $\mu s$ ) vs. number of keys ( $\times 2^{20}$ ).

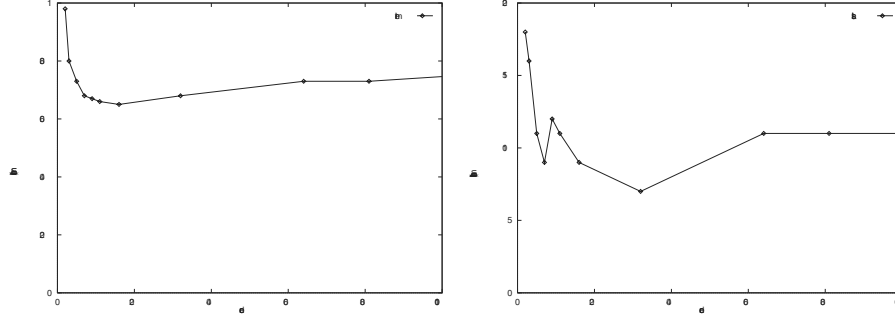


**Fig. 2.** Misses per key and instructions per key for R-MERGE and Q-MERGE (merge phase only). Misses are measured by simulation of a direct-mapped cache and do not reflect the entire memory system.

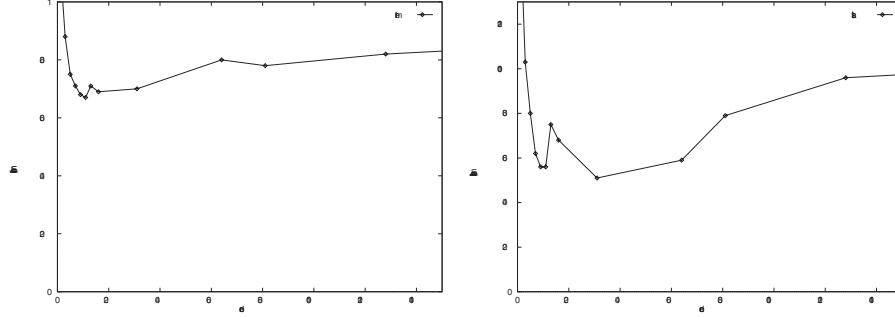
merge data structure can be accessed in registers. Q-MERGE, on the other hand, performs better with substantially larger merge order  $k$ .

Figure 1 shows performance results for R-MERGE and Q-MERGE compared with the fast implementations of [11]. We tested a range of input sizes requiring up to a gigabyte of main memory. The speedup for merge-based sorting is greater than 2; for distribution sort, 1.3. Comparing the fastest programs (quicksort and R-MERGE), R-MERGE obtains a speedup ranging from 1.1 to 1.36. Similar results were obtained on the 21264-based system.

In order to illustrate the tradeoffs between more efficient merging and a larger number of merge passes, we vary the merge order for R-MERGE and Q-MERGE. Figure 2 compares R-MERGE and Q-MERGE at varying merge order. Since these programs have identical run-formation stages, it is sufficient to examine the merge phase to compare them. R-MERGE is consistently better be-



**Fig. 3.** Time per key ( $\mu s$ ) and number of stalls (millions) sampled for R-MERGE for different merge orders.  $N = 20 \times 2^{20}$  words.



**Fig. 4.** Time per key ( $\mu s$ ) and number of stalls (millions) sampled for R-MERGE for different merge orders.  $N = 32 \times 2^{20}$  words.

cause it allocates at least part of the heap in registers. This reduces the number of instructions executed and memory locations referenced.

Figure 3 shows the time per key and number of stalls at varying merge orders  $k$ , when sorting  $20 \times 2^{20}$  words. The best performance is obtained at  $k \approx 16$ , even though the minimum stalls is obtained at  $k \approx 32$  and the number of cache misses is minimized at  $k \geq N/(BC) = 80$ .

Figure 4 shows the time per key and number of stalls at varying merge orders  $k$ , when sorting  $32 \times 2^{20}$  words. The values at  $k = 16$  and  $k = 32$  are of particular interest. When  $k \leq 16$ , the merge heap structure can be stored in registers. The time taken is closely correlated with the number of stalls, indicating that the memory system is the bottleneck. When  $k = 32$  the heap no longer fits, and the time increases even if the number of stalls decreases.

## 5 Related Work

LaMarca and Ladner [11] analyze the number of cache misses incurred by several sorting algorithms. They compare versions optimized for instructions executed with cache-conscious versions, and show that performance can be improved by

reducing cache misses even with increased instruction count. We use these algorithms as a reference point for comparing the performance of our programs.

Rahman and Raman [12] consider the slightly different problem of sorting single precision floating point numbers. They analyze the cache behaviour of Flashsort and develop a cache-efficient version obtaining a speedup of 1.1 over quicksort when sorting 32M floats. They also develop an MSB radix sort that makes use of integer operations to speed up the sorting, giving a speedup of 1.4 on uniform random data. In their subsequent paper [13], Rahman and Raman present a variant of LSB radix sort (PLSB radix sort). PLSB radix sort makes three passes over the 32-bit data, sorting 11-bit quantities at each pass. A pass consists of a presort and a global sort phase, both using counting sort. They report a speedup of roughly 2 over the sorting routines of Ladner and LaMarca [13]. However, the number of required passes for PLSB increases when working with 64-bit words rather than 32-bit words. We expect R-MERGE to be competitive with PLSB when sorting long words.

Sen and Chatterjee [16] present a model which attempts to combine cache misses and instruction count. They show that a cache-oblivious implementation of mergesort leads to inferior performance. This analysis includes the interference misses between streams of data being merged.

Sanders [14] analyzes the cache misses when accessing multiple data streams sequentially. Any access pattern to  $k = \Theta(M/B^{1+1/a})$  sequential data streams can be efficiently supported on an  $a$ -way set associative cache with capacity  $M$  and line size  $B$ . The bound is tight up to lower order terms. In addition, any number of additional accesses to a working set of size  $k \leq M/a$  does not change this bound. In our experimental setup, we have  $M = 256 \times 2^{10}$ ,  $B = 8$  and  $a = 1$ , suggesting that values of  $k$  up to several thousand may be used without significant penalty.

Sanders [15] presents a cache-efficient heap implementation called *sequence heap*. Since the sequence heap operates on  $(key, value)$  pairs, it is not possible to directly compare a heap sort based on a heap with our integer sorting program. However, in a simple test in which we equalize the data transfer by setting the key and value to half word size, our new approaches are more than three times faster than Sanders' heapsort.

Efficient register use has also been extensively studied in the architecture community [7,4,10]. Register tiling for matrix computations has been shown to give significant speedups. However, as discussed in section 3, sorting brings up different problems.

## 6 Conclusion

A key step toward developing better models is developing efficient programs for specific problems, and using them to quantify the performance effects of features not reflected in the models. We have introduced a simple cache model incorporating the principle that computation overlapped with memory access is free and then explored the effect of memory system features on sorting performance. We have formulated key principles of use in algorithm design and implementation: algorithms need to be both instruction-conscious and cache-conscious; memory

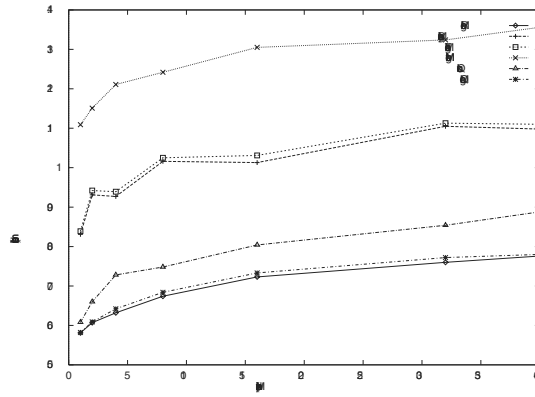
access patterns can affect performance; using registers can reduce instruction counts and memory accesses.

We illustrate these principles with a new sorting implementation called R-MERGE that is up to 36% faster than previous memory-conscious comparison sorting algorithms on our experimental system.

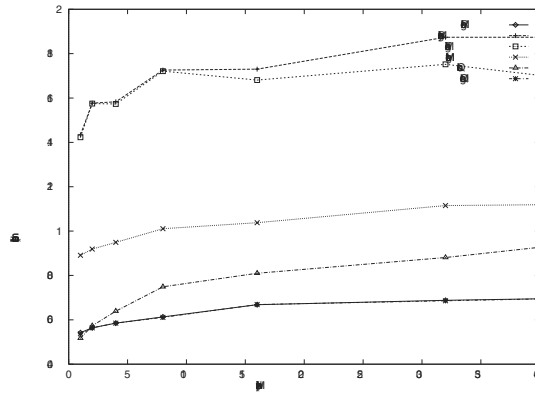
## Acknowledgements

Thanks to Andrew Gallatin, Rakesh Barve, Alvin Lebeck, Laura Toma, and SongBac Toh for their invaluable contributions. Thanks also to the many reviewers, who provided comments on the paper.

## Appendix: Other Architectures



**Fig. 5.** Results on an Intel Pentium 500MHz with 512KB cache. Time per key ( $\mu s$ ) vs. number of keys ( $\times 2^{20}$ ). Element size is 32 bits for these experiments.



**Fig. 6.** Results on an Alpha 21164A 467MHz with 2MB cache. Time per key ( $\mu s$ ) vs. number of keys ( $\times 2^{20}$ ).

## References

1. A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. *Proceedings of the 19th ACM Symposium on Theory of Computation*, pages 305–314, 1987.
2. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, M. Vandevoorde, C. Waldspurger, and B. Wehl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, October 1997.
4. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.
5. DEC. *Programmer's Guide*. Digital Unix Documentation Library. ATOM toolkit reference.
6. J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, B. J. Benschneider, D. Bernstein, R. W. Castelino, E. M. Cooper, D. E. Dever, D. R. Donchin, T. C. Fischer, A. K. Jain, S. Mehta, J. E. Meyer, R. P. Preston, V. Rajagopalan, C. Somanathan, S. A. Taylor, and G. M. Wolrich. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2 edition, 1995.
8. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
9. R. Ladner, J. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.
10. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
11. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997.
12. N. Rahman and R. Raman. Analysing cache effects in distribution sorting. In *3rd Workshop on Algorithm Engineering*, 1999.
13. N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. In *ALLENEX, Workshop on Algorithm Engineering and Experimentation*, 2000.
14. P. Sanders. Accessing multiple sequences through set associative caches. *ICALP*, 1999.
15. P. Sanders. Fast priority queues for cached memory. In *ALLENEX, Workshop on Algorithm Engineering and Experimentation*. Max-Planck-Institut für Informatik, 1999.
16. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, 2000.

# Lattice Basis Reduction with Dynamic Approximation

Werner Backes<sup>1</sup> and Susanne Wetzel<sup>2</sup>

<sup>1</sup> Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85,  
D-66123 Saarbrücken, Germany  
`backes@mpi-sb.mpg.de`

<sup>2</sup> Lucent Technologies – Bell Labs, Information Sciences Research,  
600-700 Mountain Avenue, Murray Hill, NJ 07974, USA  
`sgwetzel@research.bell-labs.com`

**Abstract.** In this paper we present a heuristic based on dynamic approximations for improving the well-known Schnorr-Euchner lattice basis reduction algorithm. In particular, the new heuristic is more efficient in reducing large problem instances and extends the applicability of the Schnorr-Euchner algorithm such that problem instances that the state-of-the-art method fails to reduce can be solved using our new technique.

## 1 Introduction

Lattices are discrete additive subgroups of the  $\mathbb{R}^n$ . Each lattice is generated by some set of linearly independent vectors, called a basis. Lattice basis reduction is a technique to construct one of the infinitely many bases of a lattice such that the basis vectors are as small as possible and as orthogonal as possible to each other. The theory of lattice basis reduction has a long history and goes back to the reduction theory of quadratic forms. It reached a high-point with the work of Lenstra, Lenstra and Lovász [15] – the well-known LLL algorithm which is the first polynomial time algorithm guaranteed to compute lattice bases consisting of relatively short vectors. Since then lattice basis reduction methods have been further developed (e.g., Schnorr-Euchner lattice basis reduction algorithm [6,22]) and proved invaluable in various areas of mathematics and computer science. In particular, the progress in lattice theory has revolutionized areas such as combinatorial optimization and cryptography (e.g., [1,2,6,12,14]). For example, lattice basis reduction techniques have been used for attacking truncated linear congruential generators [10], knapsack based hash functions [5,9] as well as cryptosystems like the Merkle-Hellman public key cryptosystem [8], RSA [7] or GGH [17].

However, despite the great progress in theory, in general lattice basis reduction methods still lack of practicability for large problem instances. Therefore, it is of great importance to further refine and improve these techniques in practice. In the following we present a new heuristic based on dynamic approximations for improving the Schnorr-Euchner lattice basis reduction algorithm which is the



most widely used lattice basis reduction algorithm in practice. In particular, the new heuristic is more efficient in reducing large problem instances and extends the applicability of the Schnorr-Euchner algorithm, i.e., problem instances that the state-of-the-art method fails to reduce can now be solved using the new technique.

The outline of the paper is as follows: At first we give a brief introduction to lattice theory. We cover the basic terminology, state basic auxiliary results and present the classical LLL algorithm as well as the Schnorr-Euchner lattice basis reduction algorithm. For further details we refer to [6,12,20]. In Section 3 we introduce the newly-developed dynamic approximation heuristic and in Section 4 we show its effectiveness on example of suitable test classes along with the corresponding test results.

## 2 Background on Lattice Basis Reduction

In the following, let  $n, k \in \mathbb{N}$  with  $k \leq n$ . By  $\|\underline{b}\|$  we denote the Euclidean length of the column vector  $\underline{b}$  and  $\lceil z \rceil$  stands for the closest integer to  $z \in \mathbb{R}$ .

An integral lattice  $L \subseteq \mathbb{Z}^n$  is defined as  $L = \left\{ \sum_{i=1}^k x_i \underline{b}_i \mid x_i \in \mathbb{Z}, i = 1, \dots, k \right\}$ , where  $\underline{b}_1, \underline{b}_2, \dots, \underline{b}_k \in \mathbb{Z}^n$  are linearly independent vectors. We call  $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$  a basis of the lattice  $L = L(B)$  with dimension  $k$ . A basis of a lattice is unique up to unimodular transformations such as exchanging two basis vectors, multiplying a basis vector by  $-1$  or adding an integral multiple of one basis vector to another one, for example.

The determinant  $\det(L) := |\det(B^T B)|^{\frac{1}{2}}$  of the lattice  $L \subseteq \mathbb{Z}^n$  with basis  $B \in \mathbb{Z}^{n \times k}$  is independent of the choice of the basis. The defect of  $B$  is defined as  $\text{dft}(B) = \frac{1}{\det(L)} \prod_{i=1}^k \|\underline{b}_i\|$ . In general,  $\text{dft}(B) \geq 1$ .  $\text{dft}(B) = 1$  iff  $B$  is an orthogonal basis. The defect can be reduced by applying lattice basis reduction methods which are techniques to construct one of the many bases of a lattice such that the basis vectors are as small as possible (by means of the Euclidean length) and are as orthogonal as possible to each other. The so-called LLL-reduction [15] is the most well-known lattice basis reduction method:

**Definition 1.** For a lattice  $L \subseteq \mathbb{Z}^n$  with basis  $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$ , corresponding Gram-Schmidt orthogonalization  $B^* = (\underline{b}_1^*, \dots, \underline{b}_k^*) \in \mathbb{Q}^{n \times k}$  and Gram-Schmidt coefficients  $\mu_{i,j}$  with  $1 \leq j < i \leq k$ , the basis  $B$  is called LLL-reduced if the following conditions are satisfied:

$$|\mu_{i,j}| \leq \frac{1}{2} \quad \text{for } 1 \leq j < i \leq k \quad (1)$$

$$\|\underline{b}_i^* + \mu_{i,i-1} \underline{b}_{i-1}^*\|^2 \geq \frac{3}{4} \|\underline{b}_{i-1}^*\|^2 \quad \text{for } 1 < i \leq k. \quad (2)$$

The first property (1) is the criterion for size-reduction (see [6,20]). The constant factor  $\frac{3}{4}$  in (2) is the so-called reduction parameter and may be replaced with any fixed real number  $y$  with  $\frac{1}{4} < y < 1$ .

The quality of an LLL-reduced lattice basis, i.e., the orthogonality and shortness of the lattice basis vectors, can be quantified by the following theorem [15]:

**Theorem 1.** *Let  $L$  be a lattice in  $\mathbb{Z}^n$  and  $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$  be an LLL-reduced basis of  $L$ . Then, the following estimates hold:*

$$\|\underline{b}_1\| \cdot \dots \cdot \|\underline{b}_k\| \leq 2^{k(k-1)/4} \det(L) \quad (3)$$

$$\|\underline{b}_1\|^2 \leq 2^{k-1} \|\underline{v}\|^2 \quad \text{for all } \underline{v} \in L, \underline{v} \neq \underline{0} \quad (4)$$

The algorithm which transforms a given lattice basis into an LLL-reduced one works as follows:

**Algorithm 1:** LLL( $\underline{b}_1, \dots, \underline{b}_k$ )

INPUT: Lattice basis  $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$

OUTPUT: LLL-reduced lattice basis

- (1) compute  $\|\underline{b}_1^*\|^2, \dots, \|\underline{b}_k^*\|^2$  and the corresponding  $\mu_{ij}$  for  $1 \leq j < i \leq k$  using the Gram-Schmidt method (c.f., [21])
- (2)  $l := 2$
- (3) **while** ( $l \leq k$ ) **do**
- (4)   REDUCE( $\mu_{ll-1}$ )
- (5)   **if** ( $\|\underline{b}_l^*\|^2 < (\frac{3}{4} - \mu_{ll-1}^2) \|\underline{b}_{l-1}^*\|^2$ ) **then**
- (6)     SWAP( $\underline{b}_{l-1}, \underline{b}_l$ )
- (7)      $l := \max\{l-1, 2\}$
- (8)   **else**
- (9)     **for** ( $m := l-2$ ;  $m \geq 1$ ;  $m--$ ) **do**
- (10)       REDUCE( $\mu_{lm}$ )
- (11)     **od**
- (12)      $l := l+1$
- (13)   **fi**
- (14) **od**

With the subroutine SWAP( $\underline{b}_{l-1}, \underline{b}_l$ ), the two basis vectors  $\underline{b}_{l-1}$  and  $\underline{b}_l$  are exchanged and REDUCE( $\mu_{lm}$ ) is used to size-reduce the element  $\mu_{lm}$ , thus achieving that  $|\mu_{l,m}| \leq \frac{1}{2}$ .

For any lattice  $L \subseteq \mathbb{Z}^n$  with basis  $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$ , the LLL-reduction of the basis  $B$  can be computed in polynomial time. More precisely, the number of arithmetic operations needed by the LLL algorithm is  $O(k^3 n \log C)$ , and the integers on which these operations are performed each have binary size  $O(k \log C)$  where  $C \in \mathbb{R}$ ,  $C \geq 2$  with  $\|\underline{b}_i\|^2 \leq C$  for  $1 \leq i \leq k$ .

Even though the algorithm performs very well from a theoretical point of view, in practice it is basically useless due to the slowness of the subroutines for the exact long integer arithmetic which has to be used in order to guarantee that no errors occur in the basis (thus changing the lattice). The first major improvement of this situation (thus allowing the use of the LLL reduction method

in practice) was achieved by Schnorr and Euchner [22] by rewriting the original LLL algorithm in such a way that approximations of the integer lattice with a faster floating point arithmetic are used for the computations of and with the Gram-Schmidt coefficients  $\mu_{i,j}$  ( $1 \leq j < i \leq k$ ) while all the other operations are done on the integer lattice using an exact integer arithmetic. In addition, heuristics for avoiding and correcting floating point errors have been introduced, thus providing a practical and reasonably fast floating point variant of the original LLL algorithm with good stability:

**Algorithm 2:** Schnorr-Euchner( $\underline{b}_1, \dots, \underline{b}_k$ )

INPUT: Lattice basis  $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$

OUTPUT: LLL-reduced lattice basis

```

(1)   $l := 2$ 
(2)   $F_c := false$ 
(3)   $F_r := false$ 
(4)  APPROX_BASIS( $B', B$ )
(5)  while ( $l \leq k$ ) do
(6)    for ( $m := 2; m \leq l; m++$ ) do
(7)      ORTHOGONALIZE( $m$ )
(8)    od
(9)    for ( $m := l - 1; m \geq 1; m--$ ) do
(10)     REDUCE( $\mu_{lm}$ )
(11)   od
(12)   if ( $F_r = true$ ) then
(13)     APPROX_VECTOR( $\underline{b}', \underline{b}$ );
(14)      $F_r := false$ 
(15)   fi
(16)   if ( $F_c = true$ ) then
(17)      $l := \max\{l - 1, 2\}$ 
(18)      $F_c := false$ 
(19)   else
(20)     if ( $B_l < (\frac{3}{4} - \mu_{ll-1}^2)B_{l-1}$ ) then
(21)       SWAP( $\underline{b}_{l-1}, \underline{b}_l$ )
(22)        $l := \max\{l - 1, 2\}$ 
(23)     else
(24)        $l := l + 1$ 
(25)     fi
(26)   fi
(27) od

```

With the procedures APPROX\_BASIS and APPROX\_VECTOR, the exact lattice basis  $B$  respectively a basis vector  $b$  are approximated by  $B'$ , respectively  $b'$ , using a floating point arithmetic, i.e., an approximate but fast data type (see [22,23] for further details). For stability reasons, in the main algorithm as well as

in both the procedures ORTHOGONALIZE (for computing the Gram-Schmidt orthogonalization) and REDUCE (to compute the size-reduction) heuristics for minimizing floating point errors [22] are applied:

**Algorithm 3:** ORTHOGONALIZE( $i$ )

INPUT:  $i$  with  $1 \leq i \leq k$  and base vectors  $\underline{b}_1, \dots, \underline{b}_i$  as well as  $\underline{b}'_1 \dots \underline{b}'_i$ ,  $\mu_{pq}$   
 for  $1 \leq q < p < i$  and  $B_j = \|\underline{b}'_j\|^2$  for  $1 \leq j < i$   
 OUTPUT:  $\mu_{i1}, \dots, \mu_{ii}$  and  $B_i = \|\underline{b}'_i\|^2$

- (1)  $B_i = \|\underline{b}'_i\|^2$
- (2) **for** ( $j := 1$ ;  $j \leq i - 1$ ;  $j++$ ) **do**
- (3)   **if** ( $|\langle \underline{b}'_i, \underline{b}'_j \rangle| < 2^{\frac{\tau}{2}} \|\underline{b}'_i\| \|\underline{b}'_j\|$ ) **then**
- (4)      $s := APPROX\_VALUE(\langle \underline{b}_i, \underline{b}_j \rangle)$
- (5)   **else**
- (6)      $s = \langle \underline{b}'_i, \underline{b}'_j \rangle$
- (7)   **fi**
- (8)    $\mu_{ij} = (s - \sum_{p=1}^{j-1} \mu_{jp} \mu_{ip} B_p) / B_j$
- (9)    $B_i = B_i - \mu_{ij}^2 B_j$
- (10) **od**
- (11)  $\mu_{ii} = 1$

**Algorithm 4:** REDUCE( $\mu_{ij}$ )

INPUT:  $\mu_{ij}, \mu_{iq}$  and  $\mu_{jq}$  for  $1 \leq q \leq j$  ( $\mu_{jj} = 1$ )  
 OUTPUT:  $\underline{b}_i$  such that  $|\mu_{ij}| \leq \frac{1}{2}$  as well as updated  $\mu_{iq}$  for  $1 \leq q \leq j$  and the boolean variables  $F_c$  and  $F_r$

- (1) **if** ( $|\mu_{ij}| > \frac{1}{2}$ ) **then**
- (2)    $F_r := true$
- (3)   **if** ( $\lceil \mu_{ij} \rceil > 2^{\frac{\tau}{2}}$ ) **then**
- (4)      $F_c := true$
- (5)   **fi**
- (6)    $\underline{b}_i := \underline{b}_i - \lceil \mu_{ij} \rceil \underline{b}_j$
- (7)   **for** ( $q := 1$ ;  $q \leq j$ ;  $q++$ ) **do**
- (8)      $\mu_{iq} := \mu_{iq} - \lceil \mu_{ij} \rceil \mu_{jq}$
- (9)   **od**
- (10) **fi**

Apparently, both the run time and the stability of the Schnorr-Euchner algorithm strongly depend on the precision of the approximations used. While high precision approximations imply a major loss in efficiency, the algorithm might run into cycles and thus not terminate due to (accumulated) floating point errors if the precision is too small. Consequently, the algorithm still lacks of efficiency in particular for large lattice bases or bases with large entries because high precision approximations have to be used. To overcome this problem, we will in the

following present a new heuristic that allows dynamic adaption of the floating point precision thus decreasing the run time of the reduction process considerably.

### 3 New Heuristic

The key idea of the new heuristic to prevent problems caused by the necessity of using high precision approximations is to work with the original lattice basis for the exact representation (i.e., apply the unimodular transformations to the original lattice basis  $B$ ) while the approximations are done in respect to  $\tilde{B} = \frac{1}{r} \cdot B$  instead of  $B$  itself where  $r$  has to be chosen skillfully. The feasibility of the heuristic is based on the following theorem:

**Theorem 2.** *For a lattice  $L \subseteq \mathbb{Z}^n$  with basis  $B = (b_1, \dots, b_k) \in \mathbb{Z}^{n \times k}$ , a lattice  $\tilde{L} = \frac{1}{r} \cdot L \subseteq \mathbb{Q}^n$  with basis  $\tilde{B} = (\tilde{b}_1, \dots, \tilde{b}_k) = \frac{1}{r} \cdot B \in \mathbb{Q}^{n \times k}$  and  $r \in \mathbb{Q}, r \neq 0$  the following holds:*

$$LLL(\tilde{B}) = \frac{1}{r} \cdot LLL(B) \quad (5)$$

*Proof.* Since  $\tilde{B}^* = \frac{1}{r} \cdot B^*$  holds for the orthogonalization and  $\tilde{\mu}_{i,j} = \mu_{i,j}$  (Gram-Schmidt coefficients) for  $1 \leq j < i \leq k$ , with Definition 1 it follows that  $LLL(\tilde{B}) = \frac{1}{r} \cdot LLL(B)$ .

Since the size of the elements to be approximated changes in the course of the reduction process, it is not possible to use the same value  $r$  for the whole reduction process but rather  $r$  has to be changed dynamically over time in order to prevent cancellations and inaccuracies.

Thus, before approximating the basis at the beginning of the reduction algorithm (see Algorithm 2), first the maximum absolute entry  $m_0 = \max \{|b_{ij}| \text{ for } 1 \leq i \leq k, 1 \leq j \leq n\}$  of the lattice basis is computed which is then used to determine a suitable value for  $r = 2^{fact}$ . Heuristically,  $fact$  was chosen such that  $\frac{m_0}{2^{fact}}$  can be fit into the data type used for the approximations without having the approximated elements become too small. Moreover, whereas APPROX\_BASIS, APPROX\_VECTOR and APPROX\_VALUE are simple data type conversions in the original Schnorr-Euchner algorithm (see Section 2), for the purpose of dynamic approximations they will have to be adjusted as follows:

**Algorithm 5:** APPROX\_BASIS( $B', B$ )

INPUT:  $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$  exact basis  
 OUTPUT:  $B' = (\underline{b}'_1, \dots, \underline{b}'_k) \in \mathbb{Q}^{n \times k}$  approximated basis

```

(1) for ( $1 \leq i \leq k$ ) do
(2)   for ( $1 \leq j \leq n$ ) do
(3)      $\underline{b}'_{ij} = (\frac{1}{2^{fact}} \underline{b}_{ij})'$ 
(4)   od
(5) od
```

As before,  $\underline{b}'_{ij}$  stands for the approximated value and  $(\frac{1}{2^{fact}} \underline{b}_{ij})'$  denotes the type conversion of  $\frac{1}{2^{fact}} \underline{b}_{ij}$ .

**Algorithm 6:** APPROX\_VECTOR( $\underline{b}'_i, \underline{b}_i$ )

INPUT:  $\underline{b}_i$  exact vector

OUTPUT:  $\underline{b}'_i$  approximated vector

```

(1) for ( $1 \leq j \leq n$ ) do
(2)    $\underline{b}'_{ij} = (\frac{1}{2^{fact}} \underline{b}_{ij})'$ 
(3) od
(4)  $m_i = \max \{ |\underline{b}_{i1}|, \dots, |\underline{b}_{in}| \}$ 
(5)  $m_0 = \max \{ m_1, \dots, m_k \}$ 
(6) if  $((m_0 < av) \text{ and } (fact > 0))$  then
(7)    $fact = fact - dv$ 
(8)   if  $(fact < 0)$  then
(9)      $fact = 0$ 
(10)  fi
(11)  APPROX_BASIS( $B', B$ )
(12) fi
```

The values  $av$  (adjust\_value) and  $dv$  (decrease\_value) also depend on the approximating data type. They are used to dynamically adjust the value  $r = 2^{fact}$  and are heuristically determined such that approximated elements do not become too small thus causing cancellations and inaccuracies. (Using **doubles** for the approximations,  $av$  and  $dv$  have heuristically be determined as  $av = 1E + 52$  and  $dv = 40$ .)

**Algorithm 7:** APPROX\_VALUE( $s', s$ )

INPUT:  $s$  exact value

OUTPUT:  $s'$  approximate value

```

(1)  $s' = (\frac{1}{2^{2 \cdot fact}} s)'$ 
```

In the following we will present test results to show the efficiency of the newly-developed heuristic. Using the original Schnorr-Euchner algorithm, these problem instances could either not be LLL-reduced or the reduction could only be performed with major computational effort.

## 4 Tests

For our tests we are using transformed knapsack lattice bases as well as transformed random lattice bases. The problem of reducing so-called knapsack lattice bases originally arises in the context of solving the knapsack problem (also known as the subset sum problem):

For given positive integers  $a_1, \dots, a_n$  (the weights) and  $S \in \mathbb{N}$  (the sum) find variables  $x_1, \dots, x_n \in \{0, 1\}$  such that

$$S = \sum_{i=1}^n x_i a_i. \quad (6)$$

In [8], the problem of solving the subset sum problem is reduced to the problem of finding a short vector in an  $(n+1)$ -dimensional lattice  $L(B) \subseteq \mathbb{Z}^{n+3}$  (later referred to as knapsack lattice) where

$$B = (b_1, \dots, b_{n+1}) = \begin{pmatrix} 2 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 2 & 0 & \cdots & 0 & 1 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 2 & 0 & 1 \\ 0 & 0 & \cdots & 0 & 2 & 1 \\ a_1 W & a_2 W & \cdots & a_{n-1} W & a_n W & SW \\ 0 & 0 & \cdots & 0 & 0 & -1 \\ W & W & \cdots & W & W & \frac{n}{2} W \end{pmatrix}. \quad (7)$$

As can be seen in [22], a short lattice vector of certain knapsack lattices can efficiently be determined by using lattice reduction methods.

Random  $n \times n$  lattices  $L \subseteq \mathbb{Z}^n$  are determined by randomly choosing each entry of the corresponding lattice basis. Since any random lattice basis is already significantly reduced [3,23], lattice basis reduction methods can be applied to compute a short lattice vector of these lattices efficiently.

Recently, various public key cryptosystems based on lattice problems have been introduced (c.f. [11]), where the private key is an “easy to reduce lattice basis” and the public key is a (unimodular) transformation of the private key, such that the corresponding basis of the same lattice cannot be reduced efficiently using current state-of-the-art lattice reduction methods. Therefore, we will show the efficiency of our newly-developed heuristic on knapsack lattice basis and random lattice bases which have been transformed using such unimodular transformations thus yielding lattice bases which are considered to be hard problem instances.

For our tests we have generated bases of knapsack lattices for fixed  $n$  and bit length  $l_1$  by choosing the weights  $a_i$  randomly from  $(0, 2^{l_1})$  and the  $x_i$  ( $1 \leq i \leq n$ ) randomly from  $\{0, 1\}$  such that exactly  $\frac{n}{2}$  of the  $x_i$ ’s are equal to one. The sum  $S$  can then easily be computed.  $W$  is taken as  $W = \lceil \sqrt{n} \rceil + 1$  (see [22]).

The random lattice bases were generated by choosing the basis entries  $b_{i,j}$  ( $1 \leq i, j \leq n$ ) randomly from  $(-2^{l_1}, 2^{l_1})$  such that the corresponding random lattices  $L(B)$  have a small determinant  $\det(L) = \Delta$  by using a modification of the LiDIA function `randomize_with_det` [16].

In both cases (i.e., for knapsack lattice bases as well as random lattice bases), the bases  $B$  are then multiplied by a unimodular transformation matrix  $T$  (i.e.,  $\det(T) = \pm 1$  and  $t_{i,j} \in \mathbb{Z}$  for  $1 \leq i, j \leq n+1$ ) whose entries are taken from

$(0, 2^{l_2})$ . The resulting lattice bases  $BT$  are dense matrices that are hard to reduce for large  $l_2$ .

For  $l_1 = 50, l_2 = 500$  (knapsack lattice bases) and  $l_1 = 6, l_2 = 750, \Delta \leq 10$  (random lattice bases) with  $n = 10, 15, 20, \dots, 80$ , we have tested the newly-developed dynamic approximation heuristic (which will be integrated into the computer algebra library LiDIA [4,16] in the future) against the currently available implementation of the original Schnorr-Euchner algorithm in LiDIA (where **bigfloats** have to be used for the approximations due to the size of the entries) and the implementation of the Schnorr-Euchner algorithm LLL\_XD in the computer algebra library NTL-3.7a [19] (which uses a very special data type **xdoubles**). The tests were performed on a Celeron 333 with 128 MB RAM. The test results are summarized in Table 1 and Table 2.

**Table 1.** Knapsack lattices (run times in seconds)

n	LiDIA		NTL
	dynamic	original	
10	0.55	8.55	1.26
15	2.52	40.61	5.81
20	7.96	140.21	17.88
25	21.33	346.30	46.56
30	44.42	669.94	94.84
35	79.90	1169.01	166.76
40	170.63	2220.61	296.03
45	308.26	3520.75	433.16
50	469.63	5990.40	725.66
55	715.63	–	1041.32
60	1076.72	–	1544.34
65	1413.17	–	2122.26
70	2050.99	–	3012.49
75	2710.73	–	3973.70
80	3937.12	–	5585.05

It can be seen that the new algorithm implementing the dynamic approximation heuristic (thus allowing approximations with **doubles**) outperforms the currently available implementations of the Schnorr-Euchner algorithm in LiDIA (requiring approximations by means of **bigfloats**) as well as the one in NTL considerably. (“–” denotes a run time  $> 6000$  seconds.) However, the improvement of the newly-developed heuristic over the LLL\_XD variant of NTL is less than the one over the bigfloat variant in LiDIA because of the special **xdouble** arithmetic implementation in NTL which provides an extended exponent in comparison to the **double** arithmetic in LiDIA. From [3,23] we know, that the size of the exponent of the approximations has great influence on the performance of the reduction algorithm. An additional increase of the efficiency of the new



**Table 2.** Random lattices (run times in seconds)

n	LiDIA		NTL
	dynamic	original	
10	0.62	2.97	1.36
15	3.91	24.57	8.33
20	12.28	95.58	27.64
25	37.52	315.35	75.94
30	93.27	850.52	178.49
35	207.95	1729.28	368.06
40	383.21	3217.82	724.24
45	666.51	5314.27	1301.74
50	1194.00	–	1882.48
55	1841.73	–	2848.07
60	2860.98	–	4303.77
65	4251.42	–	6376.87
70	6055.41	–	8749.09
75	8399.36	–	11951.20
80	11193.18	–	16219.43

heuristic can be achieved by further fine tuning the parameters  $dv$  and  $av$  (see Algorithm 6).

## 5 Conclusion

In this paper we have presented a new heuristic for the original Schnorr-Euchner algorithm which allows a major decrease of the run time. Consequently, in practice these lattice basis reduction methods are applicable to a much larger set of lattice bases and will thus also have some major impact in other areas where lattice basis reduction methods are used as tools, e.g., algorithmic number theory [6] or cryptanalysis [17,18].

## References

1. Ajtai, M.: *Generating Hard Instances of Lattice Problems*. Proceedings of the 28th ACM Symposium on Theory of Computing, pp. 99–108 (1996).
2. Ajtai, M., and Dwork, C.: *A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence*. Proceedings of the 29th ACM Symposium on Theory of Computing, pp. 284–293 (1997).
3. Backes, W., and Wetzel, S.: *New Results on Lattice Basis Reduction in Practice*. Proceedings of Fourth Algorithmic Number Theory Symposium (ANTS IV), Springer Lecture Notes in Computer Science LNCS 1838, pp. 135–152 (2000).
4. Biehl, I., Buchmann, J., and Papanikolaou, T.: *LiDIA: A Library for Computational Number Theory*. Technical Report 03/95, SFB 124, Universität des Saarlandes, Saarbrücken, Germany (1995).

5. Camion, P., and Patarin, J.: *The Knapsack Hash Function Proposed at CRYPTO '89 can be Broken*. Proceedings of EUROCRYPT '91, Springer Lecture Notes in Computer Science LNCS 547, pp. 39–53 (1991).
6. Cohen, H.: *A Course in Computational Algebraic Number Theory*. Second Edition, Springer-Verlag, Heidelberg (1993).
7. Coppersmith, D.: *Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities*. Journal of Cryptology, Vol. 10(4), pp. 233–260 (1997).
8. Coster, M.J., Joux, A., LaMacchia, B.A., Odlyzko, A.M., Schnorr, C.P., and Stern, J.: *Improved Low-Density Subset Sum Algorithms*. Journal of Computational Complexity, Vol. 2, pp. 111–128 (1992).
9. Damgård, I.B.: *A Design Principle for Hash Functions*. Proceedings of CRYPTO '89, Springer Lecture Notes in Computer Science LNCS 435, pp. 416–427 (1989).
10. Frieze, A.M., Hastad, J., Kannan, R., Lagarias, J.C., and Shamir, A.: *Reconstructing Truncated Integer Variables Satisfying Linear Congruences*. SIAM J. Comput. 17(2), pp. 262–280 (1988).
11. Goldreich, O., Goldwasser, S., and Halevi, S.: *Public-Key-Cryptosystems from Lattice Reduction Problems*. Proceedings of CRYPTO '97, Springer Lecture Notes in Computer Science LNCS 1294, pp. 112–131 (1997).
12. Grötschel, M., Lovász, L., and Schrijver, A.: *Geometric Algorithms and Combinatorial Optimization*. Second Edition, Springer-Verlag, Heidelberg (1993).
13. Joux, A., and Stern, J.: *Lattice Reduction: A Toolbox for the Cryptanalyst*. Journal of Cryptology, Vol. 11, No. 3, pp. 161–185 (1998).
14. Lagarias, J.C.: *Point Lattices*. Handbook of Combinatorics, Vol. 1, Chapter 19, Elsevier (1995).
15. Lenstra, A.K., Lenstra, H.W., and Lovász, L.: *Factoring Polynomials with Rational Coefficients*. Math. Ann. 261, pp. 515–534 (1982).
16. LiDIA Group: *LiDIA Manual*. Universität des Saarlandes/TU Darmstadt, Germany, see LiDIA homepage: <http://www.informatik.tu-darmstadt.de/TI/LiDIA> (1999).
17. Nguyen, P.: *Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto '97*. Proceedings of Crypto '99, Springer Lecture Notes in Computer Science LNCS 1666, pp. 288–304 (1999).
18. Nguyen, P., and Stern, J.: *Cryptanalysis of a Fast Public Key Cryptosystem Presented at SAC '97*. Proceedings of Selected Areas in Cryptography '98, Springer Lecture Notes in Computer Science LNCS 1556 (1999).
19. NTL homepage: <http://www.cs.wisc.edu/~shoup/ntl> (2000).
20. Pohst, M.E., and Zassenhaus, H.J.: *Algorithmic Algebraic Number Theory*. Cambridge University Press (1989).
21. Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P.: *Numerical Recipes in C*. Cambridge University Press (1995).
22. Schnorr, C.P., and Euchner, M.: *Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems*. Proceedings of Fundamentals of Computation Theory '91, Springer Lecture Notes in Computer Science LNCS 529, pp. 68–85 (1991).
23. Wetzel, S.: *Lattice Basis Reduction Algorithms and their Applications*. PhD Thesis, Universität des Saarlandes, Saarbrücken, Germany (1998).

# Clustering Data without Prior Knowledge

Javed Aslam\*, Alain Leblanc\*\*, and Clifford Stein\*\*\*

Department of Computer Science  
Dartmouth College  
Hanover, NH 03755

**Abstract.** In this paper we present a new approach for clustering a data set for which the only information available is a similarity measure between every pair of elements. The objective is to partition the set into disjoint subsets such that two elements assigned to the same subset are more likely to have a high similarity measure than elements assigned to different subsets. The algorithm makes no assumption about the size or number of clusters, or of any constraint in the similarity measure. The algorithm relies on very simple operations. The running time is dominated by matrix multiplication, and in some cases curve-fitting. We will present experimental results from various implementations of this method.

## 1 Introduction

The problem of clustering a data set into subsets, each containing similar data, has been very well studied. Several extensive studies on this topic have been written, including those of Everitt [4], Kaufman and Rousseeuw [6], and Mirkin [8]. In the typical clustering problem we are given a data set  $S$  on which a similarity measure  $\xi$  is defined, and we want to partition  $S$  into a collection of subsets  $C_1 \dots C_k$  such that two elements from the same subset  $C_i$  are more likely to have a higher similarity measure than two elements from different subsets.

We can represent such data by an undirected weighted graph where the vertices represent the data and the edge weights are determined by  $\xi$ . Letting  $\omega(a, b)$  denote the weight of edge  $ab$ , a graph can then be stored in a matrix  $M$  where  $M_{ab} = \omega(a, b)$  (given that we label the edges  $0, 1, 2 \dots n - 1$  for a graph with  $n$  nodes), and  $M_a$  is the vector formed by row  $a$ .

In this paper we present a new approach for clustering a data set. Our objective is to develop an algorithm with the following properties:

1. The only assumption we make about the weight function is that  $0 \leq \omega(a, b) \leq 1$  where a low value of  $\omega(a, b)$  denotes a smaller similarity between  $a$  and  $b$ .

---

\* This work partially supported by NSF Grants EIA-98-02068 and BCS-99-78116.

\*\* Research supported by NSF Career Award CCR-9624828, a Dartmouth Fellowship, and NSF Grant EIA-98002068.

\*\*\* Research partially supported by NSF Career Award CCR-9624828, NSF Grant EIA-98-02068, a Dartmouth Fellowship, and an Alfred P. Sloane Foundation Fellowship.

2. The data can be noisy. That is, many pairs of elements that belong to different clusters can have a high similarity measure, and many pairs of elements belonging to the same cluster can have a low similarity measure.
3. We make no assumptions about the size and number of clusters. The algorithm will determine these values based on the data.
4. There is no human intervention between the time the program reads the data and the time the output is given.

Despite the extensive literature on clustering, we found very few references about clustering algorithms that respect the first three properties. The most interesting work that came to our attention was the paper by Ben-Dor et al [2] which presents an algorithm to cluster gene expression data. However it is not clear how their algorithm performs when the average weights of clusters is not known beforehand, and how it will handle clusters with different average weights.

A typical example of an algorithm not respecting the first property is one which expects data to be mapped into a geometric space and uses the distance between two data points to compute the similarity. The same class of algorithms would not respect the second property, since geometric constraints often eliminate the possibility of having a lot of noise. If  $a$  and  $b$  are close together, and so are  $b$  and  $c$ , then  $a$  and  $c$  must also be near each other.

The third property is not followed by classical algorithms such as the k-means algorithm [4] or the self-organizing maps [7], unless the algorithm is part of a procedure which iterates on the number of clusters and uses an optimization function to determine which parameters produce the best clustering. These algorithms may not respect the fourth property either, since in many cases a human will set the parameters and select the best clustering based on his or her knowledge of the field.

*Our Approach.* In this paper we present an algorithm that satisfies the first three objectives. It generates a small set of possible solutions, of which very few (typically 3 or 4) are serious candidates. In a forthcoming paper we will present a method to extract an optimal clustering from these potential solutions, thereby satisfying the fourth property.

Our approach is based on the simple observation that although  $\omega(a, b)$  in itself may not be very reliable in deciding whether  $a$  and  $b$  should belong to the same cluster, the sets  $\{\omega(a, x) | x \in S\}$  and  $\{\omega(b, x) | x \in S\}$  can help in making this decision. More specifically, even if the process that computed the similarity between  $a$  and  $b$  produced a wrong measure, we still expect that for a sufficient number of elements  $x$  in  $S$ ,  $\omega(a, x)$  and  $\omega(b, x)$  are an accurate representation of the similarity between  $a$  and  $x$  and between  $b$  and  $x$ . For instance, if  $\omega(a, x)$  and  $\omega(b, x)$  are close for most  $x$ 's, then we can infer that  $a$  and  $b$  are likely to belong to the same cluster, regardless of  $\omega(a, b)$ . This approach, which replaces dependence on any one edge by dependence on a group of edges, is why our algorithm performs well on properties 1 and 2.

We use a simple transformation to reweight the edges of a similarity graph based on this idea and which has the following property. Assume that the edges

between vertices belonging to a same cluster (we will call them *intra-cluster* edges) are drawn from a distribution  $D_1(\mu_1, \sigma_1)$  and the edges going across clusters (we will call them *inter-cluster* edges) are drawn from a distribution  $D_2(\mu_2, \sigma_2)$  where  $D_1$  and  $D_2$  are arbitrary distributions. (In this paper  $\mu$  and  $\sigma$  indicate the mean and standard deviation.) Then after the transformation the intra-cluster edges form a distribution which closely resembles a normal distribution  $N(\tilde{\mu}_1, \tilde{\sigma}_1)$ , and similarly the distribution of the inter-cluster edges closely resembles a normal  $N(\tilde{\mu}_2, \tilde{\sigma}_2)$  where  $\tilde{\mu}_1 > \tilde{\mu}_2$ . In addition it is usually the case that  $(\tilde{\mu}_1 - \tilde{\mu}_2)/\tilde{\sigma}_2 > (\mu_1 - \mu_2)/\sigma_2$ . Therefore, after the transformation we have a better idea of how the edges are distributed, even if we do not know the parameters of the distributions. Also the increase in the ratio of the difference of the means to the standard deviation of the inter-cluster edges provides better separation between the two sets of edges.

This leads to an algorithm where at its core is a series of iterations which can be summarized by the following three steps:

1. Reweight the edges resulting from the previous iteration.
2. Select a subset of edges to threshold (or set their weight to 0). By using our knowledge about the types of distributions that will be formed, we hope to threshold a much higher proportion of inter-cluster edges.
3. Set the edge weights to restart the next iteration.

As the number of iterations increases there will be fewer and fewer inter-cluster edges remaining due to thresholding, until there are none left. At that point if there are enough intra-cluster edges left we can then identify the clusters by computing the connected components of the graph.

Although the idea behind each of these operations is simple, their exact implementation is not. We will present one alternative for step 1, three possibilities for step 2, and 4 scenarios for step 3. In addition, step 2 can be performed either by looking at all the edges at once, or working on the edges adjacent to each vertex individually. In the first case we say that we operate *globally*. In the second case we operate *by vertex*.

## 2 The Algorithm

We now present the algorithm. The pseudo-code is given in Figure 1. The input is a similarity matrix  $M$  where  $M_{ij} = \omega(i, j)$ . The objective is that  $M^{(t)}$  will contain more intra-cluster edges and fewer inter-cluster edges than  $M^{(t-1)}$  does. From  $M^{(t)}$  we compute  $C^{(t)}$ , which is a set of clusters obtained by looking at  $M^{(t)}$ . We are interested in determining whether one of the  $C^{(t)}$ 's constitutes a valid clustering of the input set. We now describe each step.

*Step 1: Computing  $M^{(t)}$  from  $M^{(t-1)}$ .* The objective in computing  $M^{(t)}$  is to better separate the intra-cluster edges from the inter-cluster edges by reassigning new edge weights. We compute  $M^{(t)}$  by setting

Input: A similarity matrix  $M^{(0)}$ , and a number of iterations  $T$ .  
 Output: A collection of clusterings  $C^{(t)}, t = 1..T$ .

```

for t = 1 to T:
  step 1: Reweight the edges in  $M^{(t-1)}$  and store them in  $M^{(t)}$ .
  step 2: For each connected component of  $M^{(t)}$ 
           Select a set of edges  $\bar{E}$  for thresholding.
           For each  $uv \in \bar{E}$ :
                $M_{uv}^{(t)} = 0$  ,  $M_{vu}^{(t)} = 0$ .
  step 2.5: Put in  $C^{(t)}$  the connected components of  $M^{(t)}$ .
  step 3: Assign weights in  $M^{(t)}$  for the edges not in  $\bar{E}$ .
  
```

Fig. 1. Pseudo-code for the algorithm.

$$M_{ij}^{(t)} = \frac{M_i^{(t-1)} M_j^{(t-1)}}{(\|M_i^{(t-1)}\|)(\|M_j^{(t-1)}\|)}$$

where  $M_i$  is the  $i$ 'th row of  $M$ .  $M_{ij}^{(t)}$  is the cosine of the angle between the vectors  $M_i^{(t-1)}$  and  $M_j^{(t-1)}$ . It is 1 when the vectors are identical and 0 when they are orthogonal. This measure is commonly used to evaluate the similarity of documents represented as vectors in the Information Retrieval community[1] [10].

*Step 2: Thresholding the Edges.* In Section 4 we will see that at the end of step 1 we expect the edge weights in  $M^{(t)}$  to form two normal distributions. The intra-cluster edges form a distribution  $N(\tilde{\mu}_1, \tilde{\sigma}_1)$ , and the inter-cluster edges form a distribution  $N(\tilde{\mu}_2, \tilde{\sigma}_2)$ . This is useful because if, for instance,  $\tilde{\mu}_1 > \tilde{\mu}_2 + 3\tilde{\sigma}_2$ , then by removing all the edges with weight lower than  $\tilde{\mu}_1$  we remove most of the inter-cluster edges while preserving half the intra-cluster edges.

Although we do not know the values of  $\tilde{\mu}_1$  and  $\tilde{\mu}_2$ , this information can still be used in order to select a set of edges  $\bar{E}$  such that  $\bar{E}$  is likely to contain a high proportion of inter-cluster edges. We will then set the weight of each edge in  $\bar{E}$  to 0. If, as a result, most of the inter-cluster edges are removed then we may be able to identify the clusters. Otherwise we use the resulting  $M^{(t)}$  (after possibly some more modification as detailed in step 3) to compute  $M^{(t+1)}$ .

Given  $M^{(t)}$  obtained after step 1 of the  $t$ 'th iteration, we have devised three strategies to select which edges go in  $\bar{E}$ . The first two methods, *histogram* and *curve-fitting*, each select a fixed thresholding value  $\pi$  based on an estimate of  $\tilde{\mu}_1$  and  $\tilde{\mu}_2$  and remove all the edges with weight smaller than  $\pi$ . In the *probabilistic* method every edge can be removed based on some probability function. Each connected component in the current  $M^{(t)}$  is processed separately.

**Histogram** When  $(\tilde{\mu}_1 - \tilde{\mu}_2)/\tilde{\sigma}_2$  is large then it may be possible to identify the peaks formed by the intra-cluster and the inter-cluster edge weight distributions because there will be relatively few intra-cluster edges around  $\tilde{\mu}_2$ , and similarly few inter-cluster edges around  $\tilde{\mu}_1$ . In the histogram method we attempt to distinguish between these two curves.

Let  $E$  be the set of edges under consideration, and let  $\mu, \sigma$  be their mean and standard deviation. To estimate the curves we build a histogram by assigning the edge weights to a predefined number  $m$  of buckets  $b_0, b_1, \dots, b_{m-1}$  so that there are enough buckets to reasonably approximate the distribution, but not so many that each bucket receives just a few edges.

Let  $n(b_i)$  be the number of edges in  $b_i$ . We can distinguish the two curves if there are three values  $u, v, w$  such that  $n(b_i)$  is mostly increasing from  $b_0$  until  $b_u$ , mostly decreasing until  $b_v$ , increasing again until  $b_w$ , and finally decreasing again until  $b_{m-1}$  with  $n(b_v)$  significantly smaller than  $n(b_u)$  and  $n(b_w)$ . Here we say “mostly” increasing and decreasing to allow for some small increase or decrease between the summit at  $u$  and  $w$  and the valley at  $v$ . We derived empirically the optimal values to use.

**Curve fitting** In this method we try to fit the edge weights to the weighted average of two normal distributions. We use five parameters – mean and standard deviations of the intra-cluster and inter-cluster edges, and the proportion of the edges that are intra-cluster edges. We estimate the distributions using the Levenberg-Marquardt method[9]. We first give an initial guess of the parameters to the Levenberg-Marquardt procedure, and then it iterates, trying to minimize the  $\chi$ -squared value of the fit between the edge weights being clustered and the cumulative distribution function defined by the five parameters described above.

**Probabilistic** In the probabilistic thresholding method each edge is assigned an estimate of the probability that it is an inter-cluster edge, and the edges are then thresholded, randomly and independently, according to these probability estimates. Specifically, simple estimates for the means and standard deviations of the inter- and intra-cluster edge weight distributions are computed, and these parameters are used to determine the probability that any given edge is inter-cluster, under the assumption that the distributions are normal. Edges are then thresholded, randomly and independently, with these probabilities. Details are provided in the full paper.

*Step 2.5: Computing  $C^{(t)}$ .* In this step we determine a set of clusters by computing the connected components of the graph generated at step 2. (See [3] for an introduction to graph algorithms.) When we threshold by vertex the resulting graph  $M^{(t)}$  is not necessarily symmetric since edge  $ij$  can be thresholded while  $ji$  is not. In this case we have the following three choices:

1. Do not make  $M^{(t)}$  symmetric after the thresholding step, and compute the connected components.
2. Do not make  $M^{(t)}$  symmetric after the thresholding step, and compute the strongly connected components.
3. Make  $M^{(t)}$  symmetric by thresholding an edge  $ij$  if  $ji$  was thresholded, and compute the connected components.

We will use the third option, and justify this choice as follows. It is not hard to argue that the first option is likely to perform badly, since a single inter-cluster edge will result in two clusters collapsing into one. For the other two

cases we will examine the cases in which selecting one option over the other is disadvantageous, either by collapsing clusters which should be disjoint or by breaking a cluster into separate subclusters.

The drawback with the second method is that two clusters  $A$  and  $B$  can be collapsed as long as there is one edge from  $A$  to  $B$  and another one from  $B$  to  $A$ . With the third method  $A$  and  $B$  will be joined only when both  $ab$  and  $ba$  ( $a \in A$  and  $b \in B$ ) are present in  $M^{(t)}$ , which is less likely to happen if  $A$  and  $B$  are indeed disjoint clusters.

The third method can be bad if cluster  $A$  is subdivided into  $A_1, A_2$  and  $A_3$  such that there are only edges directed from  $A_1$  to  $A_2$ , from  $A_2$  to  $A_3$  and from  $A_3$  to  $A_1$ . But if this is the case it could indicate that there is some extra structure in  $A$ , and that maybe  $A$  should be seen as three separate clusters. For these reasons we have selected to make  $M^{(t)}$  symmetric after the thresholding step.

*Step 3: Preparing the Next Iteration.* After we have thresholded the edges in  $\bar{E}$  we have to set the weights of the remaining edges in order to compute clusters and start the next iteration. First, for every edge  $ij$  such that  $ji$  is in  $\bar{E}$ , we also set  $M_{ij}^{(t)}$  to 0. This will make  $M^{(t)}$  symmetric. Then for every other  $M_{ij}^{(t)}$  we have considered 4 different possibilities. Either  $M_{ij}^{(t)}$  remains unchanged or we set  $M_{ij}^{(t)} = M_{ij}^{(0)}$ . In addition, we can decide that  $M_{ij}^{(t)}$  is always set to 0 if  $ij$  or  $ji$  had been threshold in a previous iteration.

### 3 Experimental Results

In this section we describe preliminary results for different versions of our algorithm. We had three objectives with these experiments. First, we wanted to see how the process of reweighting and thresholding would separate the intra-cluster and inter-cluster edges, leading to a useful algorithm. Second, we wanted to know which of the several variations described in Section 2 would give the best results. Finally, we wanted to see how each variable (number of clusters, size of clusters, original means and variances) would influence the performance of the algorithm.

In the remainder of this section we will describe the data used and then analyze the results of the experimentations. In order to evaluate the results we need a quantitative measure to compare the clusters that we compute with what we think the clusters actually are. We will use an adaptation of the precision-recall method, which has been used for document clustering (see, for instance, [1])

#### 3.1 Precision-Recall Computation

Precision-recall computation attempts to give a quantitative evaluation of the output of a clustering algorithm. For a vertex  $v$  let  $C(v)$  be the cluster containing  $v$  that was computed by an algorithm, and  $\tilde{C}(v)$  what we expect the cluster



containing  $v$  to be, either from an experts opinion in the case of real data, or from the model in the case of random data. The precision of  $v$  is the proportion of the vertices in  $C(v)$  that should really belong in a cluster with  $v$ ,  $P(v) = (|C(v) \cap \tilde{C}(v)|)/|C(v)|$ . Conversely the recall value of  $v$  is the proportion of the vertices that should be in a cluster with  $v$  that have actually been assigned to  $C(v)$  by the algorithm  $R(v) = (|C(v) \cap \tilde{C}(v)|)/|\tilde{C}(v)|$ . For a complete set  $S$ , we define the precision of the clustering  $\Omega$  of the algorithm on the set  $S$  as  $P(\Omega) = (\sum_{v \in S} P(v))/|S|$ , and the recall as  $R(\Omega) = (\sum_{v \in S} R(v))/|S|$ .

The precision and recall values are two complementary measures, each of which can be high on clusterings that are obviously wrong. If the algorithm generates  $|S|$  singletons then the precision is always 1. If it generates one cluster with  $|S|$  vertices then the recall is always 1. To handle this problem we define an error measure  $E(\Omega) = ((1 - P(\Omega))^2 + (1 - R(\Omega))^2)/2$ , which is the mean squared error. The error is 0 when the clustering is perfect.

### 3.2 Methodology

We implemented the program as described in Figure 1. There are 24 different variations of the algorithm, and we tested them all. We first fixed the distribution parameters, and then for each set of parameters we generated a random graph based on these parameters, and ran all 24 variations of the algorithms. We will describe each variation with a 4 letter code, such as VCYO, where the first character is V if the thresholding is done vertex by vertex, and G if it is done globally. The second character is H for the histogram method, P for probabilistic, and C for curve fitting. The third character is Y or N, for whether or not the thresholded edges can be set back to their original value. The last character is O if the non-thresholded edges are set to their original value, or L if they are set to the value computed at the last iteration.

We measured the performance of a variation of the algorithm on a data set by taking the iteration  $t$  for which  $C^t$  minimizes the error value as described in Section 3.1. The data we used came from three different types of distribution: the normal, uniform, and bimodal (every edge has weight  $\mu - \sigma$  or  $\mu + \sigma$ ) distributions. In each case both the intra-cluster and inter-cluster edges had the same type of distribution but with different parameters. We generated random graphs in which each edge is assigned a weight with a value sampled from the relevant distribution.

### 3.3 Summary of Results

In the results that we present we generated at least 5 data sets for each setting of parameters, and averaged the error measure. The program executes 20 iterations. Limiting the number of iterations to 20 does not affect the analysis, at least on the data that we looked at so far. Except in very few cases the error value at the end of the iterations first decreases monotonically, then increases monotonically or reaches a steady state. The optimal value was reached before the 10th iteration in all but a few cases.

**Comparison of Methods.** Table 1 contains the results of running the 24 different variations of the algorithm on all the data, and averaging the error values. For each entry in the first 3 columns we have executed the algorithm described in the left cell on more than 150 random data sets with different parameters using the distribution indicated by the column label. In the last column we combine the results from all the distributions. By looking at the table it appears that the best methods are VCNL and VHYO, and that the histogram method is in general more robust.

Other interesting observations that we draw from the results are:

1. Thresholding vertex by vertex yields better results than thresholding globally. A possible explanation for this is that a different thresholding point is selected for the edges adjacent to each vertex. Therefore selecting a bad thresholding point will impact only a few edges at time, while a bad global thresholding point could result in removing either far too many or too few edges.
2. The histogram method works best when we always use the original edge weights to start an iteration and when we allow an edge that has been thresholded to be assigned a value later. But the other two methods perform best when we use the last weights computed and set to 0 the weight of any edge that was thresholded in a prior iteration. We have no explanation for this.

To gain a better intuition of what the error value means, consider the case where we have 6 clusters of size 50, a typical data set with which we experimented. When the precision and recall values are 1 then the error is 0. When the error is near 0.083, the average for our best algorithms, the precision-recall values from our experiments that produced an error near this value were (0.9729, 0.5956), (0.6076 0.8912), (0.5952 0.9803). If we consider an error of 0.23, which is in the middle of our table, then typical precision-recall values that produced this error were (0.6344, 0.4285), (0.5521, 0.4961) and (0.4095 0.6808).

If we have only one cluster then the recall is 1 while the precision is 0.1667, giving an error value of 0.3472. Conversely if every cluster is a singleton then the precision is 1 while the recall is 0.022, resulting in an error value of 0.4782. Therefore there is some significant difference in performance depending on which version of the algorithm we use. The best ones perform very well while the worst ones produce an error value not very far from what one would obtain by always generating one cluster containing all the vertices. This difference in performance cannot be explained by the implementation choice of one parameter alone, as attested by the fact that reusing the original edge weight at each iteration is helpful with the histogram method but not with the curve-fitting method.

**Influence of Input Parameters.** We have also looked at how the performance of the algorithm changed as we varied different input parameters. The most important parameters that can influence the performance are the type of distribution, the size of the clusters  $n$ , the number of clusters  $k$ , and the relative

**Table 1.** Comparison of the performance of the 24 different variations of the algorithm. Each entry is obtained by computing the average of at least 150 error values.

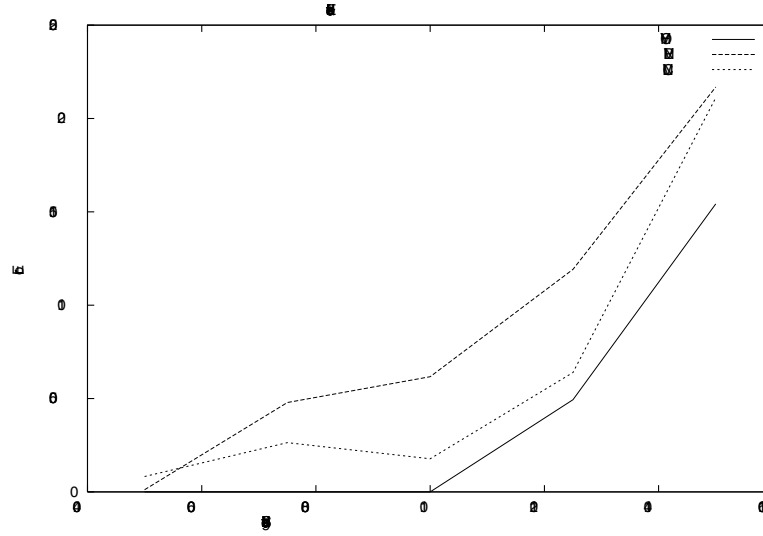
Normal		Bimodal		Uniform		Combined	
Method	Avg error	Method	Avg error	Method	Avg error	Method	Avg error
VCNL	0.084	VHYO	0.069	VHYO	0.089	VCNL	0.083
VHYO	0.090	VHYL	0.069	VCNL	0.091	VHYO	0.083
VHYL	0.095	GHYO	0.069	VHYL	0.107	VHYL	0.091
GHYO	0.101	GHYL	0.072	GHYO	0.111	GHYO	0.094
VPNL	0.106	VCNL	0.072	VPNL	0.116	GHYL	0.101
GHYL	0.110	VPNL	0.100	GHYL	0.119	VPNL	0.108
VHNL	0.119	VHNL	0.103	VHNL	0.123	VHNL	0.115
VHNO	0.137	VHNO	0.131	VHNO	0.175	VHNO	0.147
GHNO	0.231	GHNL	0.142	GHNO	0.214	GHNO	0.202
VCNO	0.232	GHNO	0.158	GHNL	0.225	GHNL	0.208
VPYL	0.235	VCYO	0.196	VPYO	0.230	GPYO	0.228
GPYO	0.238	GPYO	0.205	VCYL	0.236	VCYL	0.230
VCYL	0.239	VCYL	0.214	VPYL	0.239	VPYL	0.231
GPYL	0.241	GPYL	0.215	GPYO	0.240	VCYO	0.232
GCYL	0.245	VPYL	0.219	GPYL	0.243	GPYL	0.233
VPYO	0.247	GPNL	0.222	GPNL	0.247	VPYO	0.235
GCYO	0.247	GCNL	0.225	GCYO	0.247	GPNL	0.240
GPNL	0.249	GCYO	0.226	VCYO	0.248	GCYO	0.240
GHNL	0.250	GCYL	0.229	GCYL	0.250	GCYL	0.241
VCYO	0.253	VPYO	0.229	GCNL	0.256	GCNL	0.251
GCNL	0.272	VCNO	0.268	VCNO	0.276	VCNO	0.259
GCNO	0.298	GPNO	0.318	GPNO	0.298	GPNO	0.305
GPNO	0.298	GCNO	0.319	GCNO	0.300	GCNO	0.305
VPNO	0.307	VPNO	0.327	VPNO	0.314	VPNO	0.316

values of the intra- and inter-cluster edge weight distributions  $D_1(\mu_1, \sigma_1)$  and  $D_2(\mu_2, \sigma_2)$ . We now describe what we observed from our experiments.

The size and number of clusters have a significant impact on the performance. The performance of the algorithm will improve as the size of the clusters increases and as the number of clusters decreases. This is to be expected because in the computation of the cosine, the influence of the intra-cluster edges will diminish as the number of clusters increases. In Section 4 we will show that when all the other parameters are fixed we can always find all the clusters in one iteration, for large enough  $n$ .

The original distributions also play a major role. The most important determinant is the ratio  $\frac{\mu_1 - \mu_2}{\sigma}$  (for  $\sigma = \sigma_1$  or  $\sigma_2$ ). The performance improves as this ratio increases, since a greater value of this ratio implies fewer inter-cluster edges with a high value and/or fewer intra-cluster edges with a low value.

We have included a few plots that illustrate our results. All these plots were obtained by executing our algorithm on data that is uniformly distributed. We only included the data for the best performing variation that use each of the



**Fig. 2.** Change in error value as we increase both standard deviations which are equal. There are 6 clusters, each of size 50. The intra-cluster mean is 0.7, the inter-cluster mean is 0.6.

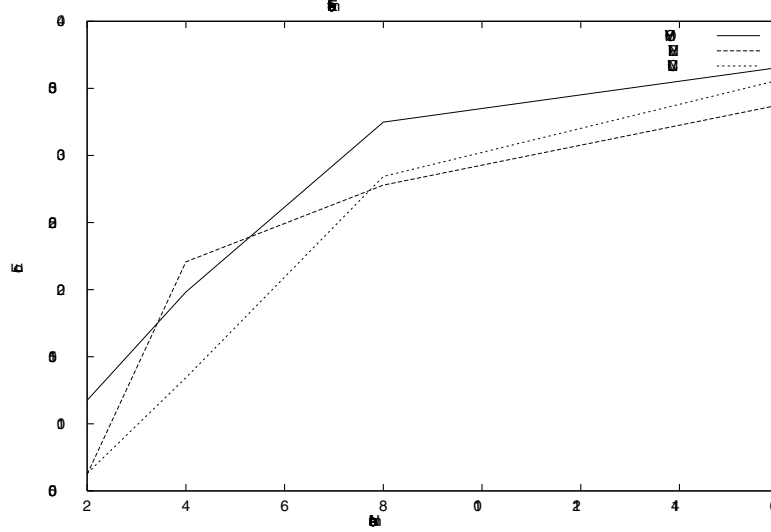
histogram, probabilistic and curve-fitting thresholding method according to table 1. In each case the error value is averaged over at least 5 different data sets. Figure 2 shows how the performance decreases as the standard deviation increases. We obtain a similar graph if we move the means closer while keeping the standard deviations fixed. Figure 3 shows the effect of increasing the number of clusters. Figure 4 illustrates how the performance increases as we increase the cluster size. Notice that in the last case each cluster has a different intra-cluster distribution.

## 4 Explanation of Results

The following analysis gives insight into why the method introduced in this paper will work. Consider the case of  $k$  clusters of size  $n$ , with two arbitrary distributions  $D_1(\mu_1, \sigma_1)$  for the intra-cluster edges and  $D_2(\mu_2, \sigma_2)$  for the inter-cluster edges. We are interested in the distributions  $\tilde{D}_1(\tilde{\mu}_1, \tilde{\sigma}_1)$  and  $\tilde{D}_2(\tilde{\mu}_2, \tilde{\sigma}_2)$  of the intra-cluster and inter-cluster edges after executing step 1 of the algorithm described in Figure 1. A key observation is the following:

**Observation 1.** *In most cases, after step 1 of the algorithm, both  $\tilde{D}_1$  and  $\tilde{D}_2$  are normally distributed, or at least very close to being normally distributed.*

To explain this phenomenon, consider a simpler version of step 1 where we compute only the dot products of every pair of vectors without normalization. Let  $D'_i(\mu'_i, \sigma'_i)$  denote the intra and inter-cluster edge distributions resulting from



**Fig. 3.** Change in error value as we vary the number of clusters. The clusters contain 30 vertices. The intra-cluster mean is 0.7, the inter-cluster mean 0.6, and both standard deviations are 0.15.

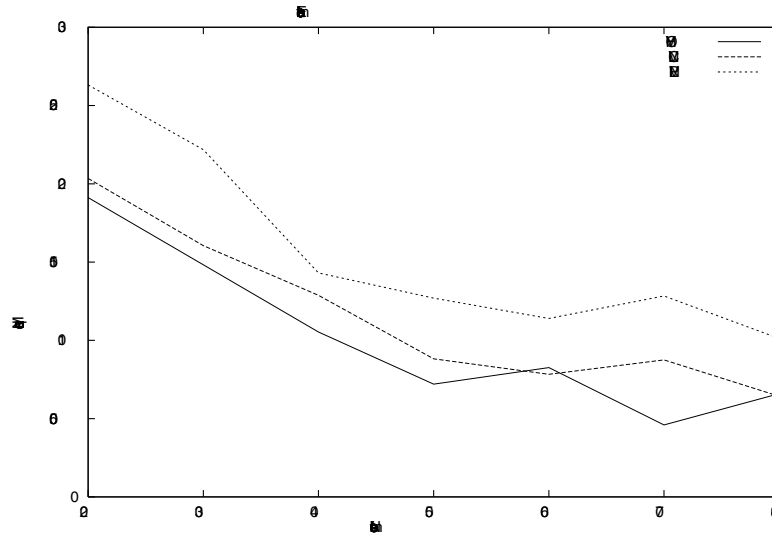
computing the dot products. Using the Central Limit theorem and basic probability theory (see for instance [5]) we observe that  $D'_1$  forms a normal distribution with mean  $n\mu_1^2 + (k-1)n\mu_2^2$  and variance  $n(\sigma_1^4 + 2\mu_1^2\sigma_1^2) + (k-1)n(\sigma_2^4 + 2\mu_2^2\sigma_2^2)$  while  $D'_2$  forms another normal distribution with mean  $2n\mu_1\mu_2 + (k-2)n\mu_2^2$  and variance  $2n(\sigma_1^2\sigma_2^2 + \mu_1^2\sigma_2^2 + \mu_2^2\sigma_1^2) + (k-2)n(\sigma_2^4 + 2\mu_2^2\sigma_2^2)$ . From this we obtain

$$\frac{\mu'_1 - \mu'_2}{\sigma'_2} = \frac{\sqrt{n}(\mu_1 - \mu_2)^2}{\sqrt{2(\sigma_1^2\sigma_2^2 + \mu_1^2\sigma_2^2 + \mu_2^2\sigma_1^2) + (k-2)(\sigma_2^4 + 2\mu_2^2\sigma_2^2)}} \quad (1)$$

The value in equation 1 indicates how far away  $\mu'_1$  is from  $\mu'_2$  as a multiple of  $\sigma'_2$ . The main objective of step 1 of the algorithm is to maximize this value. When using only the dot products  $n$  has to be huge relative to  $k$  in order for the means to be further apart after the transformation than before.

However when we normalize the vectors of the input matrix before we compute the dot product the ratio in equation 1 can be relatively big with much smaller values of  $n$ . To illustrate consider the case of  $k = 6, n = 50, \mu_1 = 0.7, \mu_2 = 0.6$ , and  $\sigma_1 = \sigma_2 = 0.1$ . After one iteration using the dot products we obtain  $(\mu'_1 - \mu'_2)/\sigma'_2 \approx 0.5$ , in which case we are worse than we were before the transformation. However if we normalize we get  $(\tilde{\mu}_1 - \tilde{\mu}_2)/\tilde{\sigma}_2 \approx 2.5$ . We can notice from Equation 1 that for  $n$  large enough, when all the other parameters are fixed, we can find all the clusters with high probability after one iteration, since the ratio  $(\mu'_1 - \mu'_2)/\sigma'_2$  grows linearly with  $\sqrt{n}$ .

We have just shown formally that  $D'_1$  and  $D'_2$  form normal distributions. We would like to be able to do the same for  $\tilde{D}_1$  and  $\tilde{D}_2$ . While it is not the



**Fig. 4.** Change in error value as we vary the size of the clusters where the intra-edge distributions vary among the clusters. The inter-cluster distribution is  $U(0.5, 0.2)$ . There are 6 clusters with distributions:  $U(0.6, 0.15)$ ,  $U(0.6, 0.15)$ ,  $U(0.7, 0.2)$ ,  $U(0.7, 0.25)$ ,  $U(0.8, 0.3)$ ,  $U(0.8, 0.35)$

case that  $\tilde{D}_1$  and  $\tilde{D}_2$  always form a normal distribution, we have significant supporting evidence to show that they are, in many cases, nearly normal. First, we have performed extensive simulations to show that the resulting distributions  $\tilde{D}_1$  and  $\tilde{D}_2$  look normal. (More details will appear in the full paper.) Second, we compute  $\tilde{D}_1$  by dividing each element of  $D'_1 = M_i M_j$  by a normalization factor  $\Theta_{ij} = \sqrt{\|M_i\| \cdot \|M_j\|}$ . Therefore each element of  $D'_1$  is divided by a different value. However the  $\Theta_{ij}$ 's form a distribution which is also close to a normal, has the same order of magnitude as  $\mu'_1$ , but which has a much smaller variance. Therefore we can approximate  $\tilde{D}_1$  by dividing everything in  $D'_1$  by a constant factor.

## 5 Conclusion and Future Work

We have presented a new method to cluster a data set based on the idea of separating the intra-cluster edges from the inter-cluster edges by analyzing the edge weight distribution. We have proposed and implemented several procedures based on this idea. We have also presented experimental results on non-trivial random data where the algorithm computes a perfect or near-perfect clustering in many cases. This method can work on data that is very noisy, and the algorithm has no prior knowledge of the expected number of clusters or of what the inter-cluster and intra-cluster distributions of the input graph are.

The next step is for the algorithm to be able to output a final set of clusters, as opposed to generate one at the end of each iteration. We have implemented an algorithm to perform this task and the first experimental results are encouraging. In addition to making the algorithm useful to cluster real data, this new algorithm should also improve on the results presented in Section 3 by combining the clusters generated through all the iterations.

On the theoretical side we would like to refine the analysis from Section 4, and better explain why some variations of the algorithm perform better.

## Acknowledgements

The authors wish to thank Bob Gross, Pablo Tamayo and Michael Angelo for helpful discussions.

## References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.
2. A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6(3/4), 1999.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
4. B. S. Everitt. *Cluster Analysis*. Oxford University Press, 1993.
5. P. G. Hoel, S. C. Port, and C. J. Stone. *Introduction to Probability Theory*. Houghton Mifflin, 1971.
6. L. Kaufman and P. J. Rousseeuw. *Finding groups in data*. John Wiley & Sons, Inc, 1990.
7. T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9), September 1990.
8. B. G. Mirkin. *Mathematical classification and clustering*. Kluwer Academic Publishers, 1996.
9. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.
10. G. Salton, A. Wong, and C. S. Yang. A vector space model for information retrieval. *Communications of the ACM*, 18(11):613–620, 1975.

# Recognizing Bundles in Time Table Graphs – A Structural Approach\*

Annegret Liebers<sup>1</sup> and Karsten Weihe<sup>2</sup>

<sup>1</sup> University of Konstanz, Department of Computer and Information Science,  
Box D 188, 78457 Konstanz, Germany  
[Annegret.Liebers@uni-konstanz.de](mailto:Annegret.Liebers@uni-konstanz.de)

<sup>2</sup> Forschungsinstitut für Diskrete Mathematik, Lennéstr. 2,  
53113 Bonn, Germany  
[weihek@acm.org](mailto:weihek@acm.org)

**Abstract.** We are dealing with an application problem arising in a co-operation with the national German railway company. It occurs in the analysis of time table data and involves inferring the underlying railroad network and the actual travel route of the trains when only their time tables are known. The structural basis of our considerations in this paper is a directed graph constructed from train time tables, where train stations correspond to vertices, and where pairs of consecutive stops of trains correspond to edges. Determining the travel route of trains amounts to an edge classification problem in this graph. Exploiting the structure of the graph, we approach the edge classification problem by locating vertices that intuitively correspond to train stations where the underlying railroad network branches into several directions, and that induce a partition of the edge set into *bundles*.

We first describe the modeling process of the classification problem resulting in the bundle recognition problem. Given the NP-hardness of the corresponding optimization problem, we then present a heuristic that makes an educated guess at an initial vertex set of potential bundle end points which is then systematically improved. Finally, we perform a computational study using time table data from 13 European countries.

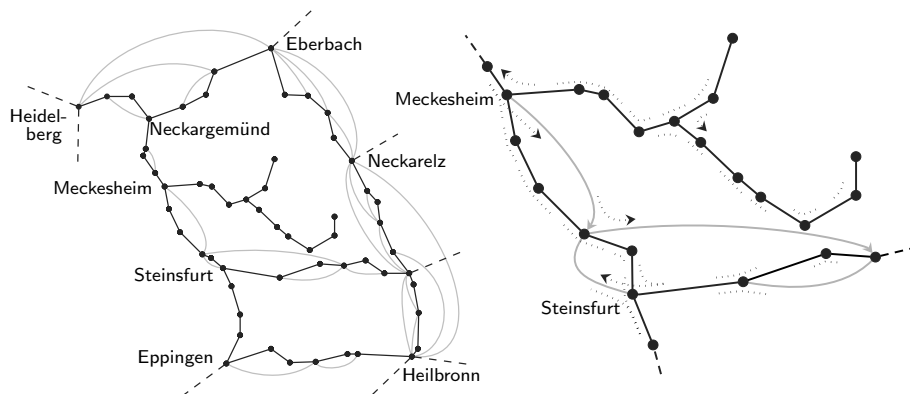
## 1 Introduction

We are given train time tables of long distance, regional, and local trains consisting of more than 140 000 trains. Together, they stop at about 28 000 train stations all over Europe. Each time table contains a list of consecutive stops for one particular train, indicating the times of arrival and departure for each stop. Such a set of train time tables induces a directed *time table graph* as follows: Each train station appearing in some time table becomes a vertex of the graph, and if there is a train leaving station  $r$  and having station  $s$  as its next stop, then there is a directed edge  $(r, s)$  in the graph. Typically, many trains contribute to one edge. So by definition, a time table graph has no multiple edges.

---

\* Partially supported by DFG grant Wa 654/10-2.





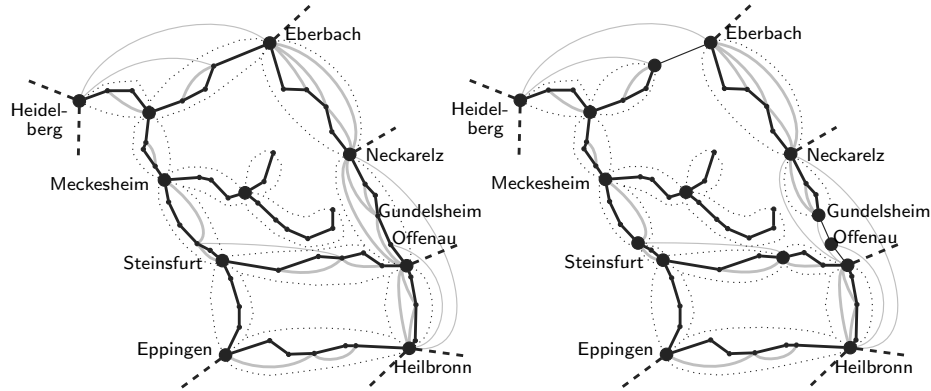
**Fig. 1.** *Left:* An example of the underlying undirected graph of a time table graph. Real edges are drawn in black, transitive edges are drawn in grey. *Right:* Line graph edges induced by the trains operating on this part of the time table graph are drawn in dotted lines; edges shown undirected mean that there is a directed edge in either direction.

The original problem as posed by the  $TLC^1$ , is the following: Decide, solely on the basis of the train time tables, for each edge of the time table graph, whether trains traveling along it pass through other train stations on the way without stopping there. If so, the sequence of these other train stations is also sought. In other words, the edges have to be classified into *real* edges representing segments of the physical railroad network and *transitive* edges corresponding to paths in this network, and for each transitive edge  $(s, t)$ , the  $s$ - $t$ -path consisting of only real edges that corresponds to the travel route of the trains contributing to  $(s, t)$  is sought. Figure 1 (left) shows an example of a time table graph with real and transitive edges.

In connection with some of their projects, the  $TLC$  is interested in an algorithmic solution for this problem based solely on train time tables. Since a European time table graph consists of more than 80 000 edges, a “manual” solution by consulting maps or other resources outside the train time tables themselves is infeasible simply because of the large size of the time table graph. After first experimenting with ad-hoc approaches using local properties such as travel times of trains between stops, we decided to tackle the edge classification problem on a structural level, and the focus of this paper is this structural approach.

Exploration of railway data suggests that, intuitively, a railroad network decomposes into lines of railroad tracks between branching points of the underlying railroad network, where the lines of railroad tracks correspond to bundles consisting of (usually many) transitive edges and one path of real edges. This is illustrated in Fig. 2 (left). Guided by this intuition, our structural approach con-

<sup>1</sup> *Transport-, Informatik- und Logistik-Consulting GmbH/EVA-Fahrplanzentrum*, the subsidiary of the national German railway company *Deutsche Bahn AG* that is responsible for collecting and publishing time table information.



**Fig. 2.** *Left:* The intuition of bundles: Branching points of the physical railroad network (big black vertices) and bundles of edges between them (encircled in dotted lines). The thin edges do not belong to any bundle. *Right:* A vertex set (big black vertices) of potential bundle end points and the edge partition resulting from applying steps 1. through 3.

sists of identifying bundles and according branching points. Section 2 illustrates the notion of bundles of edges in time table graphs. It then suggests to formulate and solve a *bundle recognition problem* in order to solve the edge classification problem. Section 3 formally defines the bundle recognition problem as a graph theoretic optimization problem by formalizing the notion of a bundle, and summarizes known NP-completeness results [2]. Section 4 describes a heuristic for bundle recognition, and Section 5 presents a computational study using train time tables from European countries, as well as an evaluation of our approach. This evaluation is the main goal of this paper.

So far, this type of problem has not been studied under an algorithmic aspect. On the other hand, some previous work addresses related yet different problems and questions: [5] discusses general problems of experimental algorithm design and refers to the problem in this paper as one of four examples. [4,3] deal with other algorithmic optimization problems on time table data, while [1] addresses visualization of edge classifications.

## 2 Modeling

The original problem consists in determining the actual travel route of trains, solely on the basis of their time tables. In discussions with our cooperation partner and in view of this and other time table analysis issues, we have developed the time table graph as an abstraction of time table data. In this graph, the original problem translates into classifying edges as real and transitive, and of finding the paths of real edges that correspond to the travel routes along transitive edges. Our approach to this edge classification problem is based on the following observation: Often there are bundle-like edge sets along a line of railroad tracks

that are delimited by the branching points of the underlying railroad network as illustrated in Fig. 2 (left). If we can find sets of edges that form such bundles, then we can, within each bundle, classify the edges: The physical railroad network appears as a (unique) Hamilton path of the bundle, and the Hamilton path immediately provides the desired paths for transitive edges. Notice that since a time table graph is a directed graph, along one line of railroad tracks there are usually two bundles, one in each direction.

In Fig. 2 (left), there are some edges (drawn in thin lines) that do not belong to any bundle. Suppose for now that the branching points and the thin edges are given. Ignoring the thin edges for a moment, the idea for finding edge sets that form bundles is then the following: If a train travels along the edge  $(r, s)$ , and then along the edge  $(s, t)$ , and if  $s$  is not a branching point, then  $(r, s)$  and  $(s, t)$  belong to the same bundle. So the following procedure would partition the edge set of the time table graph in Fig. 2 (left) into the bundles that are encircled in dotted lines:

1. Initially, every edge of the time table graph is its own set.
2. For every train and for every triple  $(r, s, t)$  of consecutive stops of the train, if  $s$  is not a big black vertex, (and if neither  $(r, s)$  nor  $(s, t)$  are thin edges,) then unite the edge sets that  $(r, s)$  and  $(s, t)$  belong to.

Each resulting bundle in Fig. 2 (left) forms a directed acyclic graph that contains a Hamilton path. Within each bundle, the edges can now be classified as real or transitive depending on whether or not they belong to the Hamilton path. But edges connecting the end points of a bundle like (Eberbach, Neckarelz) are still singletons in the partition. In most cases, we can easily assign such an edge to the bundle to which it belongs:

3. For every singleton  $(a, b)$  in the partition, if there is a unique edge set  $A'$  in the partition that contains a Hamilton path starting with  $a$  and ending with  $b$  and consisting of at least two edges, then  $(a, b)$  is a transitive edge, and the trains traveling along it actually pass through the train stations of the Hamilton path of  $A'$  without stopping there.

But in contrast to the ideal scenario described so far, in reality, we of course do not know the bundle end points of a given time table graph, nor do we know which are the edges to be ignored. Now consider Fig. 2 (right), where more vertices than just the branching points of the railroad network are marked as potential bundle end points. If we start with this vertex set and with the assumption that every edge potentially belongs to some bundle, then the edge partition resulting from steps 1. through 3. forms the edge sets encircled in dotted lines and the singletons drawn in thin lines in Fig. 2 (right). Within the encircled edge sets, edges are correctly classified, while edges in singletons remain unclassified by this approach. So by finding a vertex set of potential bundle end points (that possibly contains more vertices than just the branching points of the railroad network) such that the edge partition resulting from steps 1. through 3. consists of at least some bundles (or parts of bundles such as the one from Neckarelz to

Gundelsheim in the right part of Fig. 2) and then some singletons, we are able to classify many of the edges in a time table graph. Notice that it is edges starting or ending in the middle of a bundle like (Offenau, Heilbronn) that cause us to include more vertices than just the branching points of the railroad network in the set of potential bundle end points. As long as time table graphs do not contain too many such edges, this idea for bundle identification seems promising.

We now proceed as follows. Since singletons like (Eberbach, Neckarelz) can be assigned to their edge set with step 3. as a trivial postprocessing step after bundles (and parts of bundles) have been found with steps 1. and 2., we drop step 3. from our considerations for now and concentrate on the core of bundle recognition given by steps 1. and 2. In Section 3, we will formulate conditions that edge sets have to fulfill to constitute bundles. We seek this formalization of the intuitive notion of bundles with the following goal in mind: If a vertex set of potential bundle end points is guessed, and if steps 1. and 2. of the above procedure are applied (treating every edge as if it belongs to some bundle), then the resulting edge partition should have the following property: If the edge sets fulfill the conditions for being bundles, and if within each edge set that contains more than one element the edges are classified as real or transitive depending on whether or not they belong to the Hamilton path of the edge set, then there are no wrong classifications. Obviously, there cannot be a proof that our formalization of the notion of a bundle achieves this goal. But the visualization of the heuristic results in Fig. 5 indicates that it is a workable definition. Once the notion of a bundle has been formally defined, the bundle recognition problem then consists of finding a vertex subset of a given time table graph such that the resulting edge partition forms bundles (according to the formal definition), and such that there are few singletons in the partition.

### 3 Formalization and Known NP-Completeness Results

Recall the following well-known definition:

**Definition 1.** *Given a directed graph  $G = (V, A)$ , its line graph  $L = (A, A_L)$  is the directed graph in which every edge of  $G$  is a vertex, and in which there is an edge  $(a, b)$  if and only if there are three vertices  $r, s$ , and  $t$  of  $G$  so that  $a = (r, s)$  and  $b = (s, t)$  are edges in  $G$ .*

A set of train time tables induces the time table graph  $G = (V, A)$  together with a subset  $A'_L$  of its line graph edge set: If there is a train with train stations  $r, s$ , and  $t$  as consecutive stops, in that order, then the line graph edge  $((r, s), (s, t))$  is in  $A'_L$ . Figure 1 (right) shows an example for a subset of line graph edges in a time table graph.

**Definition 2.** *Given a directed graph  $G = (V, A)$  and a subset  $A'_L$  of its line graph edges, a vertex subset  $V' \subseteq V$  induces a partition of  $A$  by the following procedure:*

1. *Initially, every edge of  $A$  is its own set.*



**Fig. 3.** *Left:*  $V' = \{o, s\}$  would induce two edge sets, one consisting of the four short edges, and one consisting of the two long edges — for note that there are no line graph edges  $((p, q), (q, s))$  or  $((o, q), (q, r))$ . Classifying edges along Hamilton paths as real would lead to classifying all six edges as real (which is probably wrong).  $V' = \{o, q, s\}$ , however, induces four edge sets, among them two singletons, resulting in four edges classified as real and two edges remaining unclassified. *Right:*  $V' = \{o, s\}$  induces two edge sets that are opposite of each other.

2. For every line graph edge  $((r, s), (s, t)) \in A'_L$ , if  $s$  is not in  $V'$ , then unite the edge sets that  $(r, s)$  and  $(s, t)$  belong to.

We still need to specify which conditions the edge sets of a directed graph  $G = (V, A)$  induced by a line graph edge subset  $A'_L$  and a vertex subset  $V' \subseteq V$  have to fulfill to form bundles that are suitable for our edge classification approach. To begin with, such an edge set should form a directed acyclic graph that contains a Hamilton path. And since we classify edges along Hamilton paths as real, a situation as depicted in Fig. 3 (left) with  $V' = \{o, s\}$  would lead to wrongly classifying two edges as real. We can avoid such wrong classifications by demanding that two different bundles do not share vertices outside  $V'$ , unless, of course, the two bundles are *opposite* of each other as defined below and as illustrated in Fig. 3 (right). We are now ready to cast the intuitive notion of opposite edge sets into a formal definition, and to state the bundle recognition problem as a graph theoretic problem. Consider the partial order induced by the transitive closure of a directed acyclic graph:

**Definition 3.** For a directed acyclic graph  $G$ , define a partial order “ $\leq$ ” on its vertex set by  $r \leq s$  for two (not necessarily distinct) vertices of  $G$  if and only if there is a directed path (possibly of length zero) from  $r$  to  $s$  in  $G$ . If neither  $r \leq s$  nor  $s \leq r$  for two vertices  $r$  and  $s$ , we write  $r \parallel s$ .

**Definition 4.** For a directed graph  $G = (V, A)$  and a subset  $A' \subseteq A$  of its edge set, let  $G[A']$  denote the directed graph induced by  $A'$ . Given two disjoint edge subsets  $A_1 \subseteq A$  and  $A_2 \subseteq A$  of  $G$  such that  $G[A_1]$  and  $G[A_2]$  are acyclic,  $A_1$  and  $A_2$  are called *opposite* if the following two conditions hold:

1. There is an edge  $(r, s) \in A_1$  such that  $(s, r) \in A_2$ .
2. If  $r$  and  $s$  are two distinct vertices belonging both to  $G[A_1]$  and to  $G[A_2]$ , and if  $r \leq s$  in  $G[A_1]$ , then either  $s \leq r$  or  $s \parallel r$  in  $G[A_2]$ .

**Definition 5 (Bundle Recognition Problem).** Given a directed graph  $G = (V, A)$  and a subset  $A'_L$  of its line graph edge set, the bundle recognition problem consists of finding a vertex set  $V' \subseteq V$  such that the sets of the induced edge partition fulfill the following three conditions:

1. Each edge set induces a directed acyclic graph (DAG).
2. This graph contains a Hamilton path.
3. Two distinct edge sets do not have vertices outside  $V'$  in common, except possibly if the edge sets are opposite.

From now on, if a vertex set  $V' \subseteq V$  solves the bundle recognition problem, then we call the edge sets of the induced partition *bundles*. Bundles that are not singletons are called *nontrivial*. Note that a solution to the bundle recognition problem will consist in a subset  $V'$  of the vertices of the given time table graph, and that the whole vertex set  $V' := V$  of a time table graph will always constitute a solution. But this solution is useless because it induces an edge partition consisting only of singletons, and our goal is to minimize the number of induced singletons. [2] shows that finding a solution for the bundle recognition problem that contains a small number of vertices or that induces a small number of singletons is NP-hard in a very strong sense. It is already NP-complete to decide whether there is any other solution besides the trivial one:

**Theorem 1.** *Given a directed graph  $G = (V, A)$  and a subset  $A'_L$  of its line graph edge set, it is NP-complete to decide whether there is any proper subset  $V' \subset V$  solving the bundle recognition problem. It is also NP-complete to decide whether there is any vertex set  $V' \subseteq V$  solving the bundle recognition problem and inducing less than  $|A|$  singletons.*

## 4 A Heuristic for the Bundle Recognition Problem

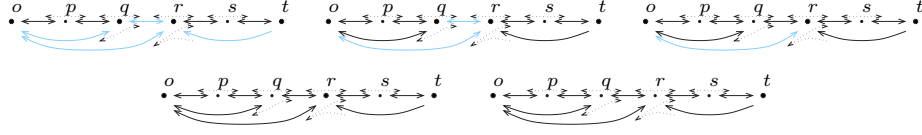
We use the fact that branching points of the physical railroad network are useful for solving the bundle recognition problem on time table graphs in practice: We make an educated guess at the set of branching points and take it as an initial vertex set  $V'$  for solving the bundle recognition problem. We then assure that wherever the edge partition induced by the initial  $V'$  does not fulfill the three bundle conditions of Def. 5, the violations of the conditions are repaired by adding more vertices to  $V'$ . Additionally to augmenting  $V'$ , we may systematically take vertices out of  $V'$  if the edge sets incident to such vertices can be united to form (larger) bundles. These ideas lead to the following heuristic:

**initial** Guess an initial set  $V' \subseteq V$  and determine the edge partition it induces.

**first augment** For each induced edge set that is not a DAG or that is a DAG but does not contain a Hamilton path, add all end vertices of its edges to  $V'$ . Also, for each vertex not in  $V'$  that is common to two non-opposite edge sets, add that vertex to  $V'$ . Determine the edge partition induced by the augmented  $V'$ .

**unite-singleton** If there is a singleton  $\{(s, t)\}$  and exactly one other bundle whose Hamilton path begins with  $s$  and ends with  $t$ , then unite the singleton with this bundle.

**reduce** If the bundles incident to a vertex  $v \in V'$  can be united to form new, larger bundles, then delete  $v$  from  $V'$  and update the edge partition accordingly. Perform *unite-singleton* and *reduce* alternately to grow bundles as



**Fig. 4.** Growing bundles by alternately applying *unite-singleton* and *reduce*: Suppose we are starting with the instance in the top left diagram with 6 vertices, 15 edges and 13 line graph edges. The initial vertex set  $V' = \{o, q, r, t\}$  (thick black dots) induces a partition with 4 nontrivial bundles (two in each direction; in black) and 7 singletons (in grey). *unite-singleton* yields the diagram in the top middle, where 3 singletons were united with nontrivial bundles. *reduce* then eliminates  $q$  from  $V'$ , leaving only two singletons (top right). Another *unite-singleton* (bottom left) and *reduce* (bottom right) yield  $V' = \{o, t\}$ , two nontrivial bundles, and no singletons for this instance.

illustrated in Fig. 4. Note that so far we do not guarantee that the resulting vertex set  $V'$  is a solution to the bundle recognition problem.

**final augment** Perform the augment step again so that the resulting vertex set  $V'$  is a solution to the bundle recognition problem.

**final unite-singleton** Perform *unite-singleton* one more time. (This is the implementation of step 3. in Section 2.)

## 5 Computational Study

The bundle recognition heuristic was implemented in C++ and tested on a set of European time tables that was made available to us by the *TLC*. Since no quality criterion for the results is available except for studying the visualization of the results with a human eye, and since this visualization suggests that our classifications are conservative in that they do not classify incorrectly but rather leave edges unclassified, calculating the percentage of classified edges is a reasonable measure. It turns out that the percentage of edges classified through bundle recognition actually indicates to what extent the bundle structure is present in different time table graphs.

As a result of the bundle recognition heuristic, 81 percent of the European time table graph edges can be classified as *real* or *transitive*. Counted separately for 13 countries, the highest percentage of classified edges (93) is reached for Slovakia, and the lowest (62) is obtained for Great Britain. The percentages for the other 11 countries are spread between 75 and 78 for 5 of them, and between 85 and 92 for the remaining 6.

In the first part of the experimental study, we tried (combinations of) the following different initial vertex sets: vertices of high degree compared to a certain percentage of that of their neighbors, vertices where trains begin or end (called *terminal vertices*), vertices that have degree at least three in one minimum spanning forest (MSF) of the (underlying undirected graph of the) time table graph (where the edge weights are either Euclidean distances or travel times along the

**Table 1.** Results of the heuristic on European time table data: Columns for Poland, the Czech Republic, Slovakia, Great Britain, Sweden, Germany, Austria, Italy, The Netherlands, Switzerland, Denmark, France and Belgium indicate the counts for vertices and edges that lie in each country, respectively. European countries not listed have less than 150 vertices in the set of time table data available for the study. Their vertices and edges only appear in the column *Eur* for the whole European time table graph. Also, edges crossing a country border are only accounted for in this column. The rows  $|V|$  and  $|A|$  list the numbers of vertices and edges within each country. Then, the results of the heuristic are given for five different choices of the initial vertex set  $V'_i$ . For each choice, the numbers of singletons (rows “s”), transitive (rows “t”) and Hamilton (rows “H”) edges in the resulting edge partition are given. Recall that when classifying edges as *real* or *transitive* using the results of the heuristic, the Hamilton edges are classified as *real*, and the singletons remain unclassified. The choice of  $V'_i$  that classified most edges is indicated in bold face for each column, and the percentage of edges classified by this best choice is given in the last row of the table.

	PL	CZ	SL	GB	S	D	A	I	NL	CH	DK	F	B	Eur
$ V $	3382	2719	919	2537	587	6505	1667	2383	375	1756	476	3304	534	28280
$ A $	8858	7369	2399	8511	1606	17793	4463	8196	1108	4586	1217	11132	1668	82594
1) Choose the initial vertex set $V'_i = V$ :														
s	987	872	212	3581	391	3115	649	2743	300	591	235	3381	462	18928
t	1216	1199	391	997	173	2159	719	1564	140	635	132	2138	254	12107
H	6655	5298	1796	3933	1042	12519	3095	3889	668	3360	850	5613	952	51559
2) $V'_i$ contains the vertices with degree larger than that of 60 % of their neighbors:														
s	939	729	150	3237	381	2642	525	1981	306	561	169	2681	433	16066
t	1250	1268	<b>419</b>	1162	177	<b>2388</b>	791	1956	138	650	160	2503	267	13555
H	6669	5372	<b>1830</b>	4112	1048	<b>12763</b>	3147	4259	664	3375	888	5948	968	52973
3) $V'_i$ contains terminal vertices and vertices that have degree at least three in the Euclidean MSF:														
s	923	759	150	3231	369	2675	513	1808	302	513	187	2557	422	15790
t	1251	1252	<b>419</b>	<b>1171</b>	<b>183</b>	2367	<b>790</b>	2048	140	<b>677</b>	154	2570	274	13695
H	6684	5358	<b>1830</b>	<b>4109</b>	<b>1054</b>	12751	<b>3160</b>	4340	666	<b>3396</b>	876	6005	972	53109
4) $V'_i$ contains terminal vertices and vertices that have degree at least three in the MSF based on travel times:														
s	906	727	150	3255	369	2677	529	1736	288	513	197	2455	412	15595
t	<b>1256</b>	<b>1266</b>	<b>419</b>	1156	<b>183</b>	2365	784	<b>2085</b>	146	<b>677</b>	146	<b>2635</b>	<b>276</b>	<b>13793</b>
H	<b>6696</b>	<b>5376</b>	<b>1830</b>	4100	<b>1054</b>	12751	3150	<b>4375</b>	674	<b>3396</b>	874	<b>6042</b>	<b>980</b>	<b>53206</b>
5) $V'_i$ contains vertices fulfilling the angle condition $\alpha_1 + \alpha_2 \leq 280^\circ$ or $\alpha_3 \geq 55^\circ$ , or having degree at least three in the Euclidean MSF:														
s	938	774	173	3300	376	2696	547	1922	270	540	169	2808	433	16283
t	1242	1245	408	1146	180	2360	780	1999	<b>154</b>	664	<b>160</b>	2446	269	13477
H	6678	5350	1818	4065	1050	12737	3136	4275	<b>684</b>	3382	<b>888</b>	5878	966	52834
%	<b>89</b>	<b>92</b>	<b>93</b>	<b>62</b>	<b>77</b>	<b>85</b>	<b>88</b>	<b>78</b>	<b>75</b>	<b>88</b>	<b>86</b>	<b>77</b>	<b>75</b>	<b>81</b>

edges), and vertices where the angles between consecutive edges in the straight line embedding of the underlying undirected graph of the time table graph indicate branching vertices. To check this *angle condition*, we determine, for each vertex  $v$  with (undirected) degree at least three, the three largest angles  $\alpha_1$ ,  $\alpha_2$ , and  $\alpha_3$  between consecutive (undirected) edges incident to  $v$ . If  $\alpha_1 + \alpha_2$  is not too large, or if  $\alpha_3$  is not too small, then we say that  $v$  fulfills the angle condition. Table 1 shows the results of the heuristic for four combinations of these choices of initial vertex sets, broken up for 13 European countries. For comparison, we also included the results when the initial vertex set consists simply of all vertices.

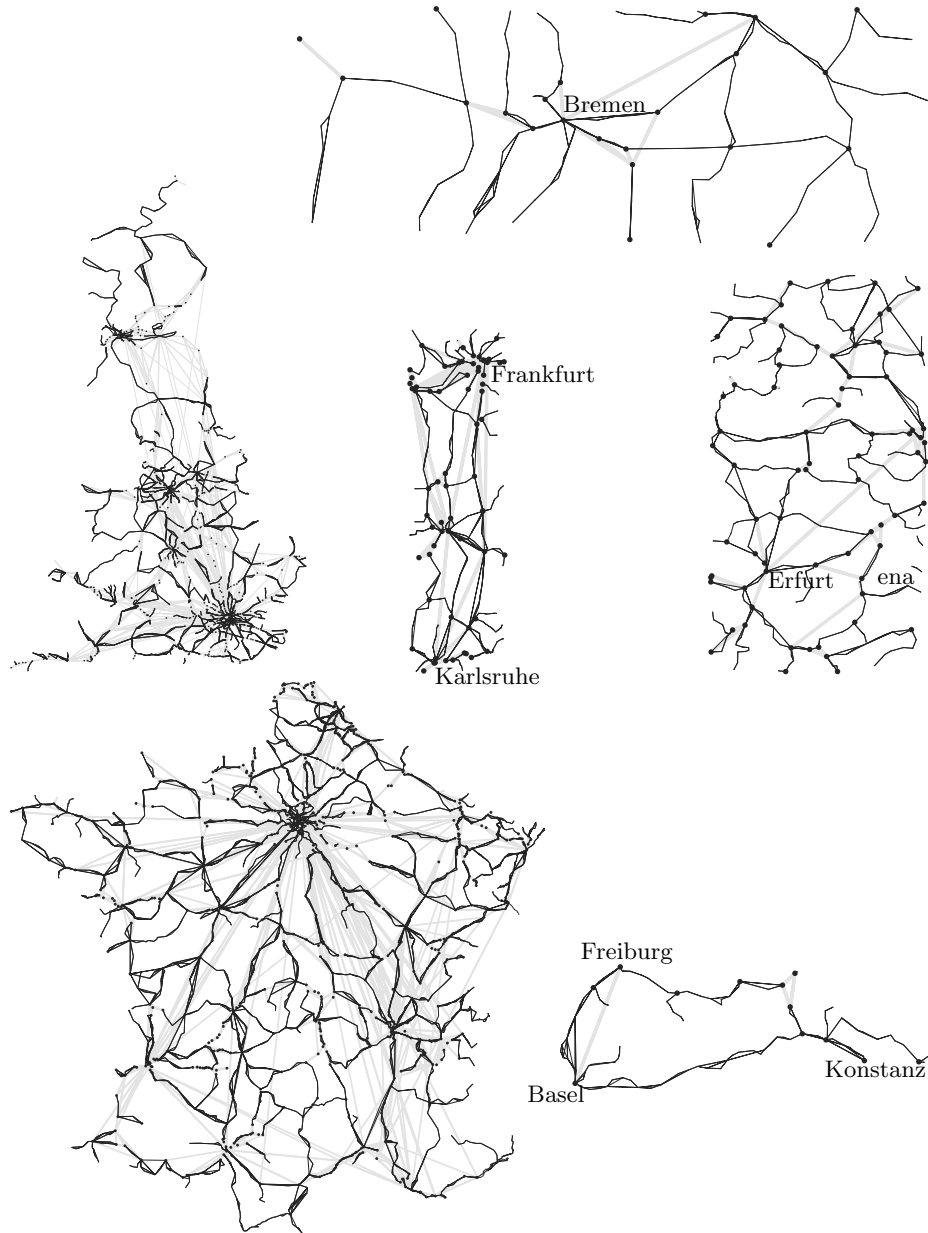


**Table 2.** Results of the heuristic for the time table graphs induced by trains traveling within Great Britain, France, and Germany, respectively. Within Germany, the results are broken down for four selected areas *sparsest*, *sparse*, *dense* and *urban* with increasing density of vertices and edges per area. The time table graphs for Great Britain and France and the time table graph excerpts for the four German areas with the edge classifications resulting from the heuristic are shown in Fig. 5. The four excerpts in Fig. 5 are drawn to scale, showing the different densities of the train network in the four German areas. For Great Britain, the initial vertex set  $V'_i$  was choice 3) from Table 1, and for France it was choice 4). For Germany, the *sparsest* and the *urban* areas,  $V'_i$  was choice 2) from Table 1. For the *sparse* and the *dense* areas, it was choice 5). It turned out that for the *sparsest*, *sparse* and *dense* areas, the result yielded by the choice of  $V'_i$  listed here was actually identical to the result for two other choices of  $V'_i$  among the five choices considered.

	GB	F	D	<i>sparsest</i>	<i>sparse</i>	<i>dense</i>	<i>urban</i>
$ V $	2 525	3 291	6 473	165	119	425	321
$ A $	8 485	10 897	16 892	404	317	1 005	906
s	3 209	2 305	1 988	18	10	78	153
t	<b>1 171</b>	<b>2 573</b>	<b>2 146</b>	<b>61</b>	<b>75</b>	<b>93</b>	<b>141</b>
H	<b>4 105</b>	<b>6 019</b>	<b>12 758</b>	<b>325</b>	<b>232</b>	<b>834</b>	<b>612</b>
%	<b>62</b>	<b>78</b>	<b>88</b>	<b>95</b>	<b>96</b>	<b>92</b>	<b>83</b>

The second part of the experimental study varies the set of time tables inducing the time table graphs considered. It is interesting to compare for different countries the results based on the European time table graph (Table 1) with the results based on the time table graph induced by the time tables of trains traveling entirely within the country (Table 2). Note that for Great Britain, France, and Germany, the best choice of initial  $V'$  for each of the three countries did not change. And comparing the results, it turns out that for each of the three countries, the extra edges in the European time table graph stemming from international trains mostly end up among the transitive and among the unclassified edges.

To evaluate our structural approach for the edge classification problem, we visualize the results for some time table graphs in Fig. 5. It shows the time table graphs of Great Britain, France, and four selected areas of Germany corresponding to Table 2 with the final vertex set  $V'$  and the recognized nontrivial bundles in black, and remaining singletons in grey. In Great Britain and France, there are many very long edges that stem from high speed and overnight trains that sometimes travel for hours without stopping anywhere in-between. Particularly in Great Britain, the percentage of such edges appears to be high, while the time table graph shows comparatively few areas with the bundle structure that the heuristic is trying to exploit. This explains why the heuristic achieves such a low percentage of classified edges in Great Britain. This lack of bundle structure contrasts sharply with the four time table graph excerpts of Germany, where the bundle structure is clearly visible, and where the heuristic works well. It recognizes (most) existing (intuitive) bundles and thus allows the classification of edges within them as *real* or *transitive*.



**Fig. 5.** Time table graphs of Great Britain and France. Excerpts of the German time table graph: *sparsest* (top), *urban* (middle), *dense* (middle right) and *sparse* (bottom right). The final vertex sets  $V'$  and the nontrivial bundles are drawn in black, while the remaining singletons are grey. The fifth pair of singletons in the *sparse* area is very short and obscured by a vertex in  $V'$ .

## 6 Conclusion

As an approach for an edge classification problem on graphs induced by train time tables, we evaluated a structural approach based on the intuition that in such graphs many edges can be partitioned into *bundles*. We conducted a computational study on time tables of 140 000 European trains. It turns out that our heuristic works quite well for time table graphs exhibiting a structure according to this intuition. The classification results based on the bundle recognition heuristic are a very good basis for further classifications using other elements of time tables besides the underlying graph structure.

## Acknowledgements

The authors would like to thank Dorothea Wagner for many fruitful discussions. They would also like to thank Stephan Schmidt, Frank Schulz, and Thomas Willhalm for software implementations that contributed to this computational study.

## References

1. Ulrik Brandes and Dorothea Wagner. Using Graph Layout to Visualize Train Interconnection Data. In *Proceedings 6th International Symposium on Graph Drawing*, pages 44–56. Springer Lecture Notes in Computer Science, vol. 1547, 1998.
2. Annegret Liebers, Dorothea Wagner, and Karsten Weihe. On the Hardness of Recognizing Bundles in Time Table Graphs. In *Proceedings 25th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 325–337. Springer Lecture Notes in Computer Science, vol. 1665, 1999.
3. Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra’s Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *Proceedings 3rd Workshop on Algorithm Engineering*, pages 110–123. Springer Lecture Notes in Computer Science, vol. 1668, 1999.
4. Karsten Weihe. Covering Trains by Stations or The Power of Data Reduction. In *On-Line Proceedings 1st Workshop on Algorithms and Experiments*, 1998. <http://rtm.science.unitn.it/alex98/proceedings.html>.
5. Karsten Weihe, Ulrik Brandes, Annegret Liebers, Matthias Müller-Hannemann, Dorothea Wagner, and Thomas Willhalm. Empirical Design of Geometric Algorithms. In *Proceedings 15th ACM Symposium on Computational Geometry*, pages 86–94, 1999.

# Analysis and Experimental Evaluation of an Innovative and Efficient Routing Protocol for Ad-hoc Mobile Networks<sup>\*</sup>

I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis

<sup>1</sup> 1. Computer Technology Institute, Patras, Greece  
{`ichatz,nikole,spirakis`}@cti.gr

<sup>2</sup> 2. Computer Engineering and Informatics Department,  
Patras University, Greece

**Abstract.** An ad-hoc mobile network is a collection of mobile hosts, with wireless communication capability, forming a temporary network without the aid of any established fixed infrastructure. In such a (dynamically changing) network it is not at all easy to avoid broadcasting (and flooding).

In this paper we propose, theoretically analyse and experimentally validate a new and efficient protocol for pairwise communication. The protocol exploits the *co-ordinated motion* of a *small* part of the network (i.e. it is a *semi-compulsory* protocol) in order to provide to various senders and receivers an efficient *support* for message passing. Our implementation platform is the LEDA system and we have tested the protocol for *three classes of graphs* (grids, random graphs and bipartite multi-stage graphs) each abstracting a different “motion topology”.

Our theoretical analysis (based on properties of *random walks*) and our experimental measurements indicate that *only a small fraction* of the mobile stations are enough to be exploited by the *support* in order to achieve *very fast communication* between any pair of mobile stations.

## 1 Introduction

Mobile computing has been introduced (mainly as a result of major technological developments) in the past few years forming a new computing environment. Because of the fact that mobile computing is constrained by poor resources, highly dynamic variable connectivity and restricted energy sources, the design of stable and efficient mobile information systems has been greatly complicated. Until now, two basic system models have been proposed for mobile computing. The “fixed backbone” mobile system model has been around the past decade and has evolved to a fairly stable system that can exploit a variety of information in order to enhance already existing services and yet provide new ones. On the other hand, the “ad hoc” system model assumes that mobile hosts can form networks without the participation of any fixed infrastructure.

---

<sup>\*</sup> This work was partially supported by the EU projects IST FET-OPEN ALCOM-FT, IMPROVING RTN ARACNE and the Greek GSRT Project PENED99-ALKAD.

An ad hoc mobile network ([9]) is a collection of mobile hosts with wireless network interfaces forming a temporary network without the aid of any established infrastructure or centralised administration. In an ad hoc network two hosts that want to communicate may not be within wireless transmission range of each other, but could communicate if other hosts between them are also participating in the ad hoc network and are willing to forward packets for them.

Suppose that  $l$  mobile hosts equipped with wireless transmitters and receivers are moving in a geographical area forming an ad hoc network. Suppose further that these hosts want to execute a simple distributed protocol such as leader election. One way to perform this is to utilise an underlying communication protocol (see [3]), which delivers (if possible) a message from one mobile host to another, regardless of their position. This scheme, in the case of high mobility of the hosts, could lead to a situation where most of the computational and battery power of the hosts is consumed for the communication protocol.

Is there a more efficient technique (other than notifying every station that the sender meets, in the hope that some of them will then eventually meet the receiver) that will effectively solve the routing problem without flooding the network and exhausting the battery and computational power of the hosts?

Routing between two hosts in a mobile ad hoc network has been a crucial problem and several approaches have been developed. Its most important performance characteristic is the amount of (routing related) traffic generated regarding the mobility rating of the hosts. Conventional routing protocols are insufficient for ad hoc networks since the routing information generated after each topology change may waste a large portion of the wireless bandwidth. In [4] propose the Dynamic Source Routing (DSR) protocol, which uses on-demand route discovery. There exist many variations of the DSR protocol that try to optimise the route discovery overhead. [2] present the AODV (Ad Hoc On Demand Distance Vector routing) protocol that also uses a demand-driven route establishment procedure. More recently TORA (Temporally-Ordered Routing Algorithm, [13]) is designed to minimise reaction to topological changes by localising routing-related messages to a small set of nodes near the change. [6] and [7] attempt to combine proactive and reactive approaches in the Zone Routing Protocol (ZRP), by initiating route discovery phase on-demand, but limit the scope of the proactive procedure only to the initiator's local neighbourhood. [11] propose the Location-Aided Routing (LAR) protocol that uses a global positioning system to provide location information that is used to improve the performance of the protocol by limiting the search for a new route in a smaller "request zone".

The innovative routing protocol that we propose here can be efficiently applied to weaker models of ad-hoc networks as no location information is needed neither for the support or for any other host of the network. Additionally, our protocol does not use conventional methods for path finding; the highly dynamic movement of the mobile hosts can make the location of a valid path inconceiv-

able - paths can become invalid immediately after they have been added to the directory tables.

Based on the work of [8] we use a graph theoretic concept where the movement of the mobile hosts in the three-dimensional space  $S$  is mapped to a graph  $G(V,E)$ ,  $|V|=n$ ,  $|E|=e$ . In particular, we abstract the environment where the stations move (in three-dimensional space with possible obstacles) by a *motion-graph* (i.e. we neglect the detailed geometric characteristics of the motion. We expect that future research will incorporate geometry constraints into the subject). In particular, we first assume (as in [8]) that each mobile host has a transmission range represented by a sphere  $tr$  centred by itself. This means that any other host inside  $tr$  can receive any message broadcasted by this host. We approximate this sphere by a cube  $tc$  with volume  $\mathcal{V}(tc)$ , where  $\mathcal{V}(tc) < \mathcal{V}(tr)$ . The size of  $tc$  can be chosen in such a way that its volume  $\mathcal{V}(tc)$  is the maximum that preserves  $\mathcal{V}(tc) < \mathcal{V}(tr)$ , and if a mobile host inside  $tc$  broadcasts a message, this message is received by any other host in  $tc$ . Given that the mobile hosts are moving in the space  $S$ ,  $S$  is divided into consecutive cubes of volume  $\mathcal{V}(tc)$ .

**Our results:** We provide a particular implementation of efficient node-to-node communication in a mobile ad-hoc network by introducing the idea of a (mobile) small-sized *support* subnetwork i.e. a subset of nodes that move in a coordinated way and act as an intermediate storage of messages. Our proposed support is a “snake-like” sequence of stations that always remains pair-wise adjacent and move in a way determined by the snake’s head. The head moves by executing a random walk on the motion graph. The protocol is implemented in the LEDA environment by extending LEDA to support *the mobile host class*, *the message class* and *the transmission medium class*. Both our measurements and our theoretical analysis indicate that *only a small support is needed for very efficient communication*. Fine-tuning of our protocol is performed via a sequence of experiments for various graphs of motion. Our theoretical analysis draws from interesting properties of random walks on graphs and their meeting times. All our implementations are written in C++ and use LEDA. The source codes are available at the following URL:

<http://helios.cti.gr/adhoc/routing.html>

## 2 The Protocol

### 2.1 The General Scheme

**Definition 1.** *The support,  $\Sigma$ , of an ad-hoc mobile network is a subset of the mobile hosts, moving in a co-ordinated way and always remaining pairwise adjacent, as indicated by the support motion subprotocol.*

**Definition 2.** *The class of ad-hoc mobile network protocols which enforce a subset of the mobile hosts to move in a certain way is called the class of semi-compulsory protocols.*

Our proposed scheme, in simple terms, works as follows: The nodes of the support move in a coordinated way so that they sweep (given some time) the entire motion graph. Their motion is accomplished in a distributed way via a *support motion subprotocol*  $P_1$ . When some node of the support is within communication range of a sender, an underlying *sensor subprotocol*  $P_2$  notifies the sender that it may send its message(s).

The messages are then stored “somewhere within the support structure”. For simplicity we may assume that they are copied and stored in every node of the support. This is not the most efficient storage scheme and can be refined in various ways. When a receiver comes within communication range of a node of the support, the receiver is notified that a message is “waiting” for him and the message is then forwarded to the receiver. For simplicity, we will also assume that message exchange between nodes within communication distance of each other takes negligible time (i.e. the messages are short packets). Note that this general scheme allows for easy implementation of many-to-one communication and also multicasting. In a way, the support  $\Sigma$  plays the role of a (moving) skeleton subnetwork (of a “fixed” structure, guaranteed by the motion subprotocol  $P_1$ ), through which all communication is routed. From the above description, the size,  $k$ , and the shape of the support may affect performance.

## 2.2 The Proposed Implementation

At the set-up phase of the ad-hoc network, a predefined number,  $k$ , of hosts, become the nodes of the support. The members of the mobile support perform a leader election, which is run once and imposes only an initial communication cost. The elected leader, denoted by  $MS_0$ , is used to co-ordinate the support topology and movement. Additionally, the leader assigns local names to the rest of the support members ( $MS_1, MS_2, \dots, MS_{k-1}$ ). The movement of  $\Sigma$  is then defined as follows:

Initially,  $MS_i, \forall i \in \{0, 1, \dots, k-1\}$ , start from the same area-node of the motion graph. The direction of movement of the leader  $MS_0$  is given by a memoryless operation that chooses randomly the direction of the next move. Before leaving the current area-node,  $MS_0$  sends a message to  $MS_1$  that states the new direction of movement.  $MS_1$  will change its direction as per instructions of  $MS_0$  and will propagate the message to  $MS_2$ . In analogy,  $MS_i$  will follow the orders of  $MS_{i-1}$  after transmitting the new directions to  $MS_{i+1}$ . Movement orders received by  $MS_i$  are positioned in a queue  $Q_i$  for sequential processing. The very first move of  $MS_i, \forall i \in \{1, 2, \dots, k-1\}$  is delayed by  $\delta$  period of time.

We assume that the mobile support hosts move with a common speed. Note that the above described motion subprotocol  $P_1$  enforces the support to move as a “snake”, with the head (the elected leader  $MS_0$ ) doing a random walk on the motion graph and each of the other nodes  $MS_i$  executing the simple protocol “move where  $MS_{i-1}$  was before”. This can be easily implemented because  $MS_i$

will move following the edge from which it received the message from  $MS_{i-1}$  and therefore our protocol does not require common sense of orientation.

In our experiments we noticed that the communication times are slightly improved when the head of the snake excludes from its random choice its current position. We also noticed that only negligible gains come by having  $MS_0$  to “remember and avoid” even many of previous snake positions.

### 2.3 Protocol Correctness

We assume here that the motions of the mobile hosts of the network which are not members of the support are determined by application protocols and that they are *independent* of the motion of the support (i.e. we exclude the case where the other hosts are deliberately trying to avoid  $\Sigma$ ). Moreover, we assume that the mobile hosts of the network have sufficient power to support motion and communication. In such a case, describing most of the reasonable networks, any particular mobile host will eventually meet some node of the support with probability 1. In fact, and by using the Borel-Cantelli Lemmas for infinite sequences of trials, given an unbounded period of (global) time (not necessarily known to the mobile stations) each station will meet the support *infinitely often* with probability 1 (since the events of meeting the support are mutually independent and the sum of their probabilities diverges). This guarantees correct delivery of a message onto the support  $\Sigma$  and, then, correct reception by a destination node when it subsequently meets the support.

### 2.4 Efficiency

Time-efficiency of semi-compulsory protocols for ad-hoc networks is not possible to estimate without a scenario for the motion of the mobile stations not in the support (the non-compulsory part). However, as in [8], we propose an “on-the-average” analysis by assuming that the movement of each mobile host  $h \notin \Sigma$  is a random walk on the graph  $G$ . We propose this kind of analysis as a necessary and interesting first step in the analysis of efficiency of any semi-compulsory or even non-compulsory protocol for ad-hoc mobile networks.

We assume further that all random walks are *concurrent* and that there is a global time  $t$ , not necessarily known to the hosts. We are now able to define the random walk of a mobile host on  $G$  that induces a continuous time Markov chain  $M_G$  as follows: The states of  $M_G$  are the vertices of  $G$ . Let  $s_t$  denote the state of  $M_G$  at time  $t$ . Given that  $s_t = u$ ,  $u \in V$ , the probability that  $s_{t+dt} = v$ ,  $v \in V$ , is  $p(u, v) \cdot dt$  where

$$p(u, v) = \begin{cases} \frac{1}{d(u)} & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}, \text{ where } d(u) \text{ is the degree of vertex } u.$$

We denote  $E_\mu [ \ ]$  the expectation for the chain started at time 0 from any vertex with distribution  $\mu$  (e.g. the initial distribution of the Markov chain).



Let  $T_i = \min\{t \geq 0 : s_t = i\}$  be the first hitting time on state  $i$  (the first time that the mobile host visits vertex  $i$ ). The total time needed for a message to arrive from a sending node  $u$  to a receiving node  $v$  is equal to:

$$T_{total} = X + T_\Sigma + Y \quad (1)$$

where:

$X$  is the time for the sender node to reach a node of the support.

$T_\Sigma$  is the time for the message to propagate inside the support. Clearly,  $T_\Sigma = \mathcal{O}(k)$ , i.e. linear in the support size.

$Y$  is the time for the receiver to meet with a support node, after the propagation of the message inside the support.

To estimate the expected values of  $X$  and  $Y$  (they are random variables), we work as follows:

(a) Note first that  $X, Y$  are, statistically, of the same distribution, under the assumption that  $u, v$  are randomly located (at the start) in  $G$ . Thus  $E(X) = E(Y)$ .

(b) We now replace the meeting time of  $u$  and  $\Sigma$  by a hitting time, using the following thought experiment: (b1) We fix the support  $\Sigma$  in an “average” place inside  $G$ . (b2) We then collapse  $\Sigma$  to a single node (by collapsing its nodes to one but keeping the incident edges). Let  $H$  be the resulting graph,  $\sigma$  the resulting node and  $d(\sigma)$  its degree. (b3) We then estimate the hitting time of  $u$  to  $\sigma$  assuming  $u$  is somewhere in  $G$ , according to the *stationary distribution*,  $\pi$ , of its walk, on  $H$ . We denote the expected value of this hitting time by  $E_\pi T_\sigma^H$ .

Thus, now  $E(T_{total}) = 2E_\pi T_\sigma^H + \mathcal{O}(k)$ .

*Note 1.* The equation above amortises over meeting times of senders (or receivers) because it uses the stationary distribution of their walks for their position when they decide to send a message.

*Note 2.* Since the motion graph  $G$  is finite and connected, the continuous chain abstracting the random walk on it is automatically time-reversible.

Now, proceeding as in [1] we have (proof in the full paper, see [17])

**Lemma 1.** *For any node  $\sigma$  of any graph  $H$  in a continuous-time random walk*

$$E_\pi T_\sigma^H \leq \frac{\tau_2(1 - \pi_\sigma)}{\pi_\sigma}$$

where  $\pi_\sigma$  is the (stationary) probability of the walk at node (state)  $\sigma$  and  $\tau_2$  is the relaxation time of the walk.

*Note 3.* In the above bound,  $\tau_2 = \frac{1}{\lambda_2}$  where  $\lambda_2$  is the second eigenvalue of the (symmetric) matrix  $S = \{s_{i,j}\}$  where  $s_{i,j} = \sqrt{\pi_i} p_{i,j} (\sqrt{\pi_i})^{-1}$  and  $P = \{p_{i,j}\}$  is the transition matrix of the walk. Since  $S$  is symmetric, it is diagonalizable and the spectral theorem gives the following representation:  $S = U\Lambda U^{-1}$  where  $U$  is

orthonormal and  $\Lambda$  is a diagonal real matrix of diagonal entries  $0 = \lambda_1 < \lambda_2 \leq \dots \leq \lambda_{n'}$ ,  $n' = n - k + 1$  (where  $n' = |V_H|$ ). These  $\lambda$ 's are the eigenvalues of both  $P$  and  $S$ . In fact,  $\lambda_2$  is an indication of the *expansion* of  $H$  and of the asymptotic rate of convergence to the stationary distribution, while relaxation time  $\tau_2$  is the corresponding interpretation measuring time.

It is a well-known fact (see e.g. [15]) that  $\forall v \in V_H$ ,  $\pi_v = \frac{d(v)}{2m'}$  where  $m' = |E_H|$  is the number of the edges of  $H$  and  $d(v)$  is the degree of  $v$  in  $H$ . Thus  $\pi_\sigma = \frac{d(\sigma)}{2m'}$ .

By estimating  $d(\sigma)$  and  $m'$  and remarking that the operation of locally collapsing a graph does not reduce its expansion capability and hence  $\lambda_2^H \geq \lambda_2^G$ .

**Theorem 1.**

$$E(X) = E(Y) \leq \frac{1}{\lambda_2(G)} \Theta\left(\frac{n}{k}\right)$$

**Theorem 2.** *The expected communication time of our scheme is bounded above by the formula*

$$E(T_{total}) \leq \frac{2}{\lambda_2(G)} \Theta\left(\frac{n}{k}\right) + \Theta(k)$$

*Note 4.* The above upper bound is minimised when  $k = \sqrt{\frac{2n}{\lambda_2(G)}}$ , a fact also verified by our experiments.

We remark that this upper bound is tight in the sense that by [1], Chapter 3,

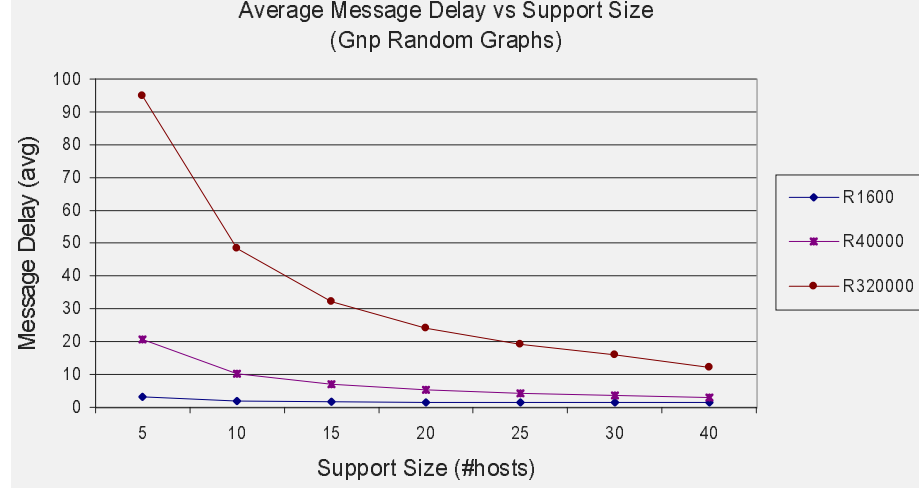
$$E_\pi T_\sigma^H \geq \frac{(1 - \pi_\sigma)^2}{q_\sigma \pi_\sigma}$$

where  $q_\sigma = \sum_{j \neq \sigma} p_{\sigma j}$  in  $H_j$  is the total exit rate from  $\sigma$  in  $H$ . By using martingale type arguments we can show sharp concentration around the mean degree of  $\sigma$ .

### 3 Algorithmic Engineering

The fine-tuning of our protocol is a subject of algorithmic engineering since the theoretical analysis gives only bounds on the communication time. In particular, the analysis cannot easily answer the question of how much helpful it is for the head of  $\Sigma$  to remember (and avoid) past positions occupied by  $\Sigma$ . Experiments however indicate that limited memory (remembering a few past positions) incurs a slight improvement on the communication times, while bigger memory is not helpful at all.

More crucially, the analysis indicates the important fact that *only a small sized support  $\Sigma$  is needed to achieve very efficient communication times*. This size (actually of order equal to the square root of the number of vertices in the motion graph) is indeed verified experimentally.



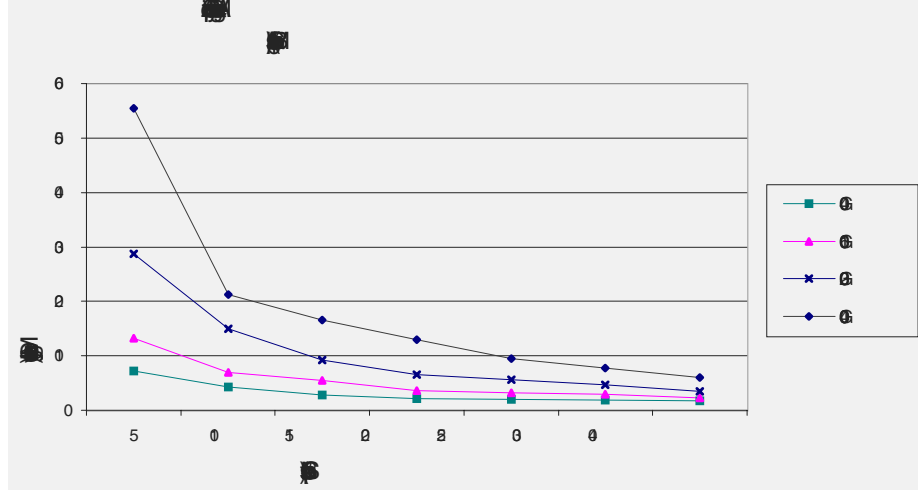
**Fig. 1.** Average message delay over Gnp types of motion graphs with different sizes  $n$  and varied support size  $k$ .

A number of experiments were carried out modelling the different possible situations regarding the geographical area covered by an ad hoc mobile network. We used one sender for generating and transmitting messages and one receiver for the destination of the messages and carried out the experiment with at least 100,000 messages exchanges regarding protocol generated traffic. In some cases (such as the bipartite multi-stage graphs) we extended the message count to 150,000.

For each message generated, we calculated the delay  $X$  (in terms of rounds) until the message has been transmitted to a member of  $\Sigma$ . Additionally, for each message received by the support, we logged the delay  $Y$  (in terms of rounds) until the message was transmitted to the receiver. We assumed that no extra delay was imposed by the mobile support hosts (i.e.  $T_{\Sigma} = 0$ ). We then used the average delay over all measured fractions of message delays for each experiment in order to compare the results and discuss them further.

*2D Grid Graphs.* Our experimentation started with simple 2D grid graphs ( $\sqrt{n} \times \sqrt{n}$ ). We used three different grid graphs with  $n \in \{100, 400, 1600\}$  over different sizes of  $\Sigma$  with  $k \in \{5, 10, 15, 20, 25, 30, 40, 60, 80, 100, 120, 160, 240, 320, 400, 480, 640\}$  (see figure 3).

*$G_{n,p}$  Random Graphs.* In order to experiment on more realistic cases, we used the  $G_{n,p}$  model of random graphs, obtained by sampling the edges of a complete graph of  $n$  nodes independently with probability  $p$ . We used  $p = 1.05n \log n$  which is marginally above the threshold value for  $G_{n,p}$  connectivity. The test cases included random graphs with  $n \in \{100, 400, 1600, 3200, 40000, 320000\}$  and  $\Sigma$  size  $k \in \{5, 10, 15, 20, 25, 30, 40\}$  (see figure 1).



**Fig. 2.** Average message delay over bipartite multi-stage types of motion graphs with different sizes ( $n$ ) and varied support size  $k$ .

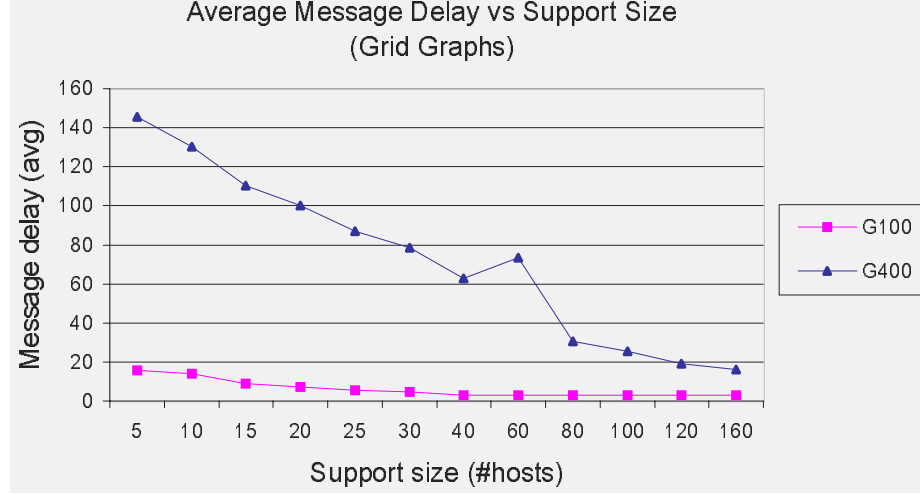
*Bipartite Multi-stage Graphs.* The last set of experiments used the bipartite multi-stage graphs or series parallel graphs. In order to construct the graphs we used  $\xi$  stages, where  $\xi$  is a parameter. We used  $p = \frac{n}{\xi} \log \frac{n}{\xi}$  which is the threshold value for connectivity of each pair of stages. The test cases included multi-stage graphs with 2, 3 or 4 stages and a total number of nodes  $n \in \{100, 400, 1600, 3200, 40000\}$ . The size of  $\Sigma$  we used was  $k \in \{5, 10, 15, 20, 25, 30, 35, 40\}$  (see figure 2).

### 3.1 Discussion on the Experiments

The experimental results have been used to compare the average message delays with the upper bound provided by the theoretical analysis of the protocol. We observe that as the total number  $n$  of motion-graph nodes remains constant, as we increase the size  $k$  of  $\Sigma$ , the total message delay (i.e.  $E(T_{total})$ ) is decreased. Actually,  $E(T_{total})$  initially decreases very fast with increasing  $k$ , while having a limiting behaviour of no further significant improvement when  $k$  crosses a certain threshold value. Therefore, taking into account a possible amount of statistical error, the following has been experimentally validated:

$$\text{if } k_1 > k_2 \Rightarrow E_1(T_{total}) < E_2(T_{total})$$

Throughout the experiments we used small and medium sized graphs, regarding the total number of nodes in the motion graph. Over the same type of graph (i.e. grid graph) we observed that the average message delay  $E(T_{total})$  increases as the total number of nodes increases. This can be clearly seen from figure 2 where the curves of the three different graph-sizes are displayed. It is safe to assume that for the same graph type and for a fixed-size  $\Sigma$  the overall



**Fig. 3.** Average message delay over grid types of motion graphs with different sizes ( $n$ ) and varied support size  $k$ .

message delay increases with the size of the motion graph. We further observe that  $E(T_{total})$  does not increase linearly but is affected by the expansion rate of the graph, or otherwise, as stated in the theoretical analysis, by the second eigenvalue of the matrix of the graph on communication times. If we take into account possible amount of statistical error that our experiments are inherently prone to, we can clearly conclude the following:

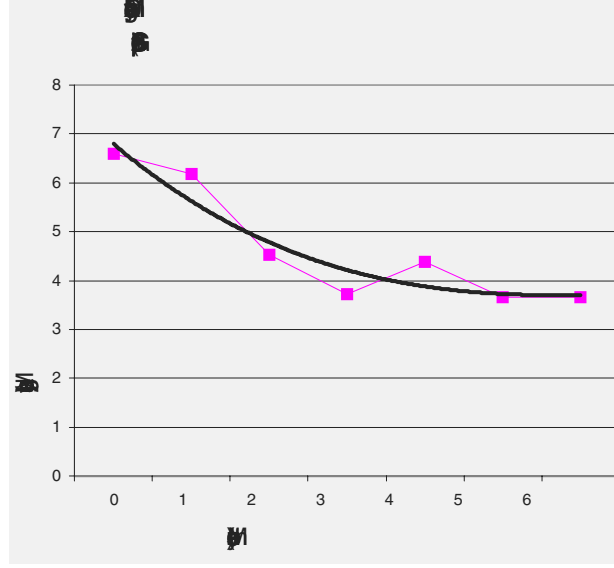
$$\text{if } n_1 > n_2 \Rightarrow E_1(T_{total}) > E_2(T_{total})$$

Furthermore, we remark that  $E(T_{total})$  does not only depend on the actual size of the graph and the size of the support  $\Sigma$  but additionally that it is directly affected by the type of the graph. This is expressed throughout the theoretical analysis by the effect of the second eigenvalue of the matrix of the graph on communication times. If we combine the experimental results of figures 1,2 and 3 for fixed number of motion-graph nodes (e.g.  $n=400$ ) and fixed number of  $\Sigma$  size (e.g.  $k=40$ ) we observe the following:

$$E_{grid}(T_{total}) > E_{multistage}(T_{total}) > E_{G_{n,p}}(T_{total})$$

which validates the corresponding results due to the analysis.

In section 2.4, we note that the upper bound on communication times  $E(T_{total}) \leq \frac{2}{\lambda_2(G)} \Theta\left(\frac{n}{k}\right) + \Theta(k)$  is minimised when  $k = \sqrt{\frac{2n}{\lambda_2(G)}}$ . Although our experiments disregard the delay imposed by  $\Sigma$  (i.e.  $\Theta(k) \rightarrow 0$ ) this does not significantly affect our measurements. Interestingly, the values of figures 1,2 and 3 imply that the analytical upper bound is not very tight. In particular, for the  $G_{n,p}$  graphs we clearly observe that the average message delay drops below 3 rounds (which is a very low, almost negligible, delay) if  $\Sigma$  size is equal to 10;



**Fig. 4.** The effect of variable memory size over the average message delay. The experiments use a grid type graph for the motion graph with a fixed number of nodes ( $n$ ) and a fixed support size  $k=20$ .

however, the above formula implies that this size should be higher, especially for the medium sized graphs. Actually, it is further observed that there exists a certain threshold value (as also implied by the theoretical analysis) for the size of  $\Sigma$  above which the overall message delay does not further significantly improve. The threshold size for the support validated experimentally is smaller than the threshold implied by the analytical bound.

The experimental work on the proposed routing protocol provides an important insight for fine-tuning of the original algorithm according to which the head of  $\Sigma$  remembers previous positions so that it does not re-visit them within a short period of time.

## 4 Future Work

We intend to strengthen our results by providing a tighter analysis on the concentration around the mean of the degree of the vertex corresponding to the collapsing of the support structure. We also wish to investigate alternative (not necessarily “snake-like”) structures for the support and comparatively study the corresponding effect of support size on communication times.

## References

1. D. Aldous and J. Fill: Reversible Markov Chains and Random Walks on Graphs. Unpublished manuscript. <http://stat-www.berkeley.edu/users/aldous/book.html> (1999).
2. C. E. Perkins and E. M. Royer: Ad hoc on demand distance vector (AODV) routing. IETF, Internet Draft, draft-ietf-manet-aodv-04.txt (IETF'99). (1999).
3. M. Adler and C. Scheideler: Efficient Communication Strategies for Ad-Hoc Wireless Networks. In Proc. 10th Annual Symposium on Parallel Algorithms and Architectures (SPAA'98)(1998).
4. J. Broch, D. B. Johnson, and D. A. Maltz: The dynamic source routing protocol for mobile ad hoc networks. IETF, Internet Draft, draft-ietf-manet-dsr-01.txt, Dec. 1998. (1998).
5. B. Das, R. Sivakumar and V. Bharghavan: Routing in ad hoc networks using a virtual backbone. In Proc. IEEE (IC3N'97) (1997).
6. Z. J. Haas and M. R. Pearlman: The performance of a new routing protocol for the reconfigurable wireless networks. In Proc. ICC '98. (1998).
7. Z. J. Haas and M. R. Pearlman: The zone routing protocol (ZRP) for ad hoc networks. IETF, Internet Draft, draft-zone-routing-protocol-02.txt, June 1999. (1999).
8. K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas and R. B. Tan: Fundamental Control Algorithms in Mobile Networks. In Proc. 11th Annual Symposium on Parallel Algorithms and Architectures (SPAA'99) (1999).
9. T. Imielinski and H. F. Korth: Mobile Computing. Kluwer Academic Publishers. (1996).
10. P. Johansson, T. Larsson, N. Hedman, B. Mielczarek and M. Degermark: Scenario-based Performance Analysis of Routing Protocols for Mobile Ad-Hoc Networks. In Proc. Mobile Computing (MOBICOM'99) (1999).
11. Y. Ko and N. H. Vaidya: Location-Aided Routing (LAR) in Mobile Ad Hoc Networks. In Proc. Mobile Computing (MOBICOM'98). (1998).
12. K. Mehlhorn and S. Naher: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press. (1999).
13. V. D. Park and M. S. Corson: Temporally-ordered routing algorithms (TORA) version 1 functional specification. IETF, Internet Draft, draft-ietf-manet-tora-spec-02.txt, Oct. 1999. (1999).
14. G. Aggelou and R. Tafazolli.: RDMAR: A Bandwidth-efficient Routing Protocol for Mobile Ad Hoc Networks. In Proc. 2nd ACM International Worksh. on Wireless Mobile Multimedia (WoWMom'99) (1999).
15. R. Motwani and P. Raghavan: Randomized Algorithms. Cambridge University Press. (1995).
16. E. M. Royer and C. K. Toh: A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks. (1999).
17. I. Chatzigiannakis, S. Nikolettseas and P. Spirakis: Analysis and Experimental Evaluation of an Innovative and Efficient Routing Protocol for Ad-hoc Mobile Networks (full paper). CTI Technical Report. Aug. 2000. <http://helios.cti.gr/adhoc/routing.html> (2000).

# Portable List Ranking: An Experimental Study

Isabelle Guérin Lassous<sup>1</sup> and Jens Gustedt<sup>2</sup>

<sup>1</sup> INRIA Rocquencourt

Isabelle.Guerin-Lassous@inria.fr

<sup>2</sup> LORIA and INRIA Lorraine

Jens.Gustedt@loria.fr

**Abstract.** We present two portable algorithms for the List Ranking Problem in the Coarse Grained Multicomputer model (CGM) and we report on experimental studies of these algorithms. With these experiments, we study the validity of the chosen CGM model, and also show the possible gains and limits of such algorithms.

## 1 Introduction and Overview

*Why List Ranking.* The *List Ranking Problem*, LRP, reflects one of the basic abilities needed for efficient treatment of dynamical data structures, namely the ability to follow arbitrarily long chains for references. Many parallel graph algorithms use List Ranking as a subroutine. But before handling graph in parallel/distributed, it is useful to know the possibilities and the limits of the LRP in a practical setting.

A linked list is a set of nodes such that each node points to another node called its successor and there is no cycle in such a list. The LRP consists in determining the rank for all nodes, that is the distance to the last node of the list. In this paper, we work in a more general setting where the list is cut into sublists. Then, the LRP consists in determining for all nodes its distance to the last node of its sublist.

Whereas this problem seems (at a first glance) to be easily tractable in a sequential setting, techniques to solve it efficiently in parallel quickly get quite involved and are neither easily implemented nor do they perform well in a practical setting in most cases. Many of these difficulties are due to the fact that until recently no general purpose model of parallel computation was available that allowed easy and portable implementations.

*Some parallel models.* The well studied variants of the LRP, see Karp and Ramachandran (1990) for an overview, are *fine grained* in nature, and written in the PRAM model; usually in algorithms for that model every processor is only responsible for a constant sized part of the data but may exchange such information with any other processor at constant cost. These assumptions are far from being realistic for a foreseeable future: the number of processors will very likely be much less than the size of the data and the cost for communication —be it



in time or for building involved hardware— will be at least proportional to the width of the communication channel.

Other studies followed the available architectures (namely interconnection networks) more closely but had the disadvantage of not carrying over to different types of networks, and then not to lead to portable code.

This gap between the available architectures and theoretical models was narrowed by Valiant (1990) by defining the so-called *bulk synchronous parallel* machine, BSP. Based upon the BSP, the model that is used in this paper, the so-called *Coarse Grained Multiprocessor*, CGM, was developed to combine theoretical abstraction with applicability to a wide range of architectures including clusters, see Dehne et al. (1996).

*Previous algorithms in the coarse grained models.* The first proposed algorithm is a randomized algorithm by Dehne and Song (1996) that performs in  $O(\log p \log^* n)$  rounds ( $p$  is the number of processors and  $n$  the size of the linked list) with a workload (total number of local steps) and total communication size of  $O(n \log^* n)$  ( $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$ ). Then, Caceres et al. (1997) gives a deterministic algorithm that needs  $O(\log p)$  rounds and a workload/total communication of  $O(n \log p)$ . We think that the obvious sequential algorithm performs so well, that care must be taken that only a small overhead arises in communication and CPU time. Moreover, as far as we know, no implementations of these algorithms have been carried out.

*Previous practical work.* Very few articles deal with the implementation sides of LRP. Reid-Miller (1994) presents a specific implementation optimized for the Cray C-90 of different PRAM algorithms that gives good results. In Dehne and Song (1996), some simulations have been done, but they only give some results on the number of communication rounds. Sibeyn (1997) and Sibeyn et al. (1999) give several algorithms for the LRP with derived PRAM techniques and new ones. They fine-tune their algorithms according to the features of the interconnection network the Intel Paragon. The results are good and promising, since more than 10 processors are used. In all these works, the implementations are specific to the interconnection network of the target machine and do not seem to be portable.

*This paper.* In this paper, we address ourselves to the problem of designing algorithms that give portable (as far as we know, it is the first portable proposed code on the subject), efficient and predictive code for LRP. We do not pretend to have the best implementation of the LRP, but we tried to respect the three goals (portable, efficient, predictive) at the same time. Especially, the implementation was done carefully to have the best possible results without losing at portability level. We propose two algorithms designed in the CGM model that have better workload and total communication size than the previously known algorithms in the coarse grained models and we give the experimental results of their implementations. Due to space limitations, we prefer to focus on the results obtained on a specific PC cluster because we think that this kind of networks is one of the major current trends in parallel/distributed computing. But our code

runs also without modifications on a PC cluster with a different interconnection network, a SGI Origin 2000, a Cray T3E and SUN workstations (that shows the portability aspect).

Table 1 compares the existing coarse grained algorithms and the algorithms we propose concerning the number of communication rounds and the total CPU time and the total size of exchanged data. *rand* stands for randomized and *det* for deterministic. The algorithm of Sibeyn (1997), not expressed in the CGM model, has different measures, but we mention it for its practical interest.

**Table 1.** Comparison of our results to previous work. *O*-notation omitted.

reference	comm. rounds	CPU time & communication	
Dehne and Song (1996)	$\log p \log^* n$	$n \log^* n$	rand
Caceres et al. (1997)	$\log p$	$n \log p$	det
we	$\log p \log^* p$	$n \log^* p$	det
we	$\log p$	$n$	rand
Sibeyn (1997)		$n$	aver

The paper is organized as follow: we give the main features of the CGM model in Section 2. Next, we present a deterministic algorithm for solving the LRP in Section 3.1, and a randomized algorithm in Section 3.2. Section 4 concerns the results of the implementations.

## 2 The CGM Model for Parallel/Distributed Computation

The CGM model initiated by Dehne et al. (1996) is a simplification of BSP proposed by Valiant (1990). These models have a common machine model: a set of processors that is interconnected by a network. A processor can be a monoprocesor machine, a processor of a multiprocessors machine or a multiprocessors machine. The network can be any communication medium between the processors (bus, shared memory, Ethernet, etc).

The CGM model describes the number of data per processor explicitly: for a problem of size  $n$ , it assumes that the processors can hold  $O(\frac{n}{p})$  data in their local memory and that  $1 \ll \frac{n}{p}$ . Usually the later requirement is put in concrete terms by assuming that  $p \leq \frac{n}{p}$  because each processor has to store information about the other processors.

The algorithms are an alternation of *supersteps*. In a superstep, a processor can send or receive once to and from each other processor and the amount of data exchanged in a superstep by one processor is at most  $O(\frac{n}{p})$ . Unlike BSP, the supersteps are not assumed to be synchronized explicitly. Such a synchronization is done implicitly during the communications steps. In CGM we have to ensure that the number  $R$  of supersteps is small compared to the size of the input. It can be shown that the *interconnection latency* which is one of the major bottlenecks for efficient parallel computation can be neglected if  $R$  is a function that only

**Algorithm 1: Jump**


---

**Input:** Set  $R$  of  $n$  linked items  $e$  with pointer  $e.\text{succ}$  and distance value  $\text{dist}$  and subset  $S$  of  $R$  of marked items.

**Task:** Modify  $e.\text{succ}$  and  $e.\text{dist}$  for all  $e \in R \setminus S$  s.t.  $e.\text{succ}$  points to the nearest element  $s \in S$  according to the list and s.t.  $e.\text{dist}$  holds the sum of the original  $\text{dist}$  values along the list up to  $s$ .

**while** there are  $e \in R \setminus S$  s.t.  $e.\text{succ} \notin S$  **do**

**for** all such  $e \in R$  **do**

**A**    **Invariant:** Every  $e \notin S$  is linked to by at most one  $f.\text{succ}$  for some  $f \in R$ .

1    Fetch  $e.\text{succ} \rightarrow \text{dist}$  and  $e.\text{succ} \rightarrow \text{succ}$ ;

2     $e.\text{dist} += e.\text{succ} \rightarrow \text{dist}$ ;

3     $e.\text{succ} = e.\text{succ} \rightarrow \text{succ}$

---

depends on  $p$  (and not on  $n$  the size of the input), see Guérin Lassous et al. (2000).

### 3 Two Coarse Grained Algorithms for List Ranking

The two proposed algorithms are based on PRAM techniques used to solve the LRP. Translate directly the PRAM algorithms into CGM algorithms would lead to algorithms with  $O(\log n)$  supersteps, what the CGM model does not recommend. Some works have to be added to ensure that not too many supersteps are realized in the CGM algorithms. For instance, we can use techniques that ensure that after  $O(\log p)$  supersteps the size of the problem is such that the problem can be solved sequentially on one processor. The idea is then to choose and to adapt the appropriate PRAM techniques in order to design CGM algorithms with a limited number of supersteps. At the same time, we have to pay attention to the workload, as the total communication bandwidth.

#### 3.1 A Deterministic Algorithm

The deterministic algorithm we propose to solve the LRP is based on two ideas given in PRAM algorithms. The first and basic technique, called *pointer jumping*, was mentioned by Wyllie (1979). The second used PRAM technique is a *k-ruling set*, see Cole and Vishkin (1989). We use *deterministic symmetry breaking* to obtain a *k-ruling set*, see Jája (1992). Such a *k-ruling set*  $S$  is a subset of the items in the list  $L$  s.t.

1. Every item  $x \in L \setminus S$  is at most  $k$  links away from some  $s \in S$ .
2. No two elements  $s, t \in S$  are neighbors in  $L$ .

Concerning the translation in the CGM model, the problem is to ensure that the size of the *k-ruling sets* decreases quickly at each recursive call to limit the number of recursive calls (and then of supersteps) and to ensure that the distance

**Algorithm 2:** ListRanking<sub>k</sub>

**Input:**  $n_0$  total number of items,  $p$  number of processors, set  $L$  of  $n$  linked items with pointer `succ` and distance value `dist`.

**Task:** For every item  $e$  set  $e.succ$  to the end of its sublist  $t$  and  $e.dist$  to the sum of the original `dist` values to  $t$ .

**if**  $n \leq n_0/p$  **then**

1 | Send all data to processor 0 and solve the problem sequentially there.

**else**

2 | Shorten all consecutive parts of the list that live on the same processor. ;

3 | **for every item**  $e$  **do**  $e.lot = \text{processor-id of } e$ ;

4 |  $S = \text{Ruling}_k(p-1, n, L)$ ;

5 |  $\text{Jump}(L, S)$ ;

6 | **for all**  $e \in S$  **do** set  $e.succ$  to the next element in  $S$  ;

7 | ListRanking<sub>k</sub>( $S$ );

8 |  $\text{Jump}(L, \{t\})$ ;

**Algorithm 3:** RuleOut

**Input:** item  $e$  with fields `lot`, `succ` and `pred`, and integers  $l_1$  and  $l_2$  that are set to the `lot` values of the predecessor and successor, resp.

**if**  $(e.lot > l_1) \wedge (e.lot > l_2)$  **then**

1 | Declare  $e$  *winner* and force  $e.succ$  and  $e.pred$  *loser*;

**else**

2 | **if**  $l_1 = -\infty$  **then** Declare  $e$  *p-winner* and suggest  $e.succ$  *s-looser* ;

3 | **else**

| Let  $b_0$  be the most significant bit, for which  $e.lot$  and  $l_1$  are distinct;

| Let  $b_1$  the value of bit  $b_0$  in  $e.lot$ ;

|  $e.lot := 2 * b_0 + b_1$ .

between two elements in the  $k$ -ruling set is not too long otherwise it would imply a consequent number of supersteps to link the elements of the  $k$ -ruling set.

Due to space limitation, we only present part of the algorithms. For the detailed explanations and the proofs of the correctness and the complexity, see Guérin Lassous and Gustedt (2000). Algorithm 2 solves the LRP in the CGM model. The whole procedure for a  $k$ -ruling set is given in Algorithm 4. The inner (and interesting) part of such a  $k$ -ruling algorithm is given in Algorithm 3 and Algorithm 1 reflects the pointer jumping technique.

**Proposition 1.** *ListRanking<sub>k</sub> can be implemented on a CGM such that it uses  $O(\lceil \log_2 p \rceil \log_2^* p)$  communication rounds and requires an overall bandwidth and processing time of  $O(n \log_2^* p)$ .*

For the proof of Proposition 1, see Guérin Lassous and Gustedt (2000).

**Algorithm 4:**  $\text{Ruling}_k$ 


---

**Constants:**  $k > 9$  integer threshold

**Input:** Integers  $q$  and  $n$ , set  $R$  of  $n$  linked items  $e$  with pointer  $e.\text{succ}$ , and integer  $e.\text{lot}$

**Output:** Subset  $S$  of the items s.t. the distance from any  $e \notin S$  to the next  $s \in S$  is  $< k$ .

**A Invariant:** Every  $e$  is linked to by at most one  $f.\text{succ}$  for some  $f \in R$ , denote it by  $e.\text{pred}$ .

**B Invariant:**  $q \geq e.\text{lot} \geq 0$ .

1  $\text{range} := q$ ;  
 repeat  
 C    **Invariant:** If  $e.\text{lot} \neq -\infty$  then  $e.\text{lot} \neq e.\text{succ} \rightarrow \text{lot}$ .  
 B'    **Invariant:** If  $e.\text{lot} \notin \{+\infty, -\infty\}$  then  $\text{range} \geq e.\text{lot} \geq 0$ .  
     **for** all  $e \in R$  that are neither winner nor loser **do**  
     2    Communicate  $e$  and the value  $e.\text{lot}$  to  $e.\text{succ}$  and receive the corresponding values  $e.\text{pred}$  and  $l_1 = e.\text{pred} \rightarrow \text{lot}$  from the predecessor of  $e$ ;  
     3    Communicate the value  $e.\text{lot}$  to  $e.\text{pred}$  and receive the value  $l_2 = e.\text{succ} \rightarrow \text{lot}$ ;  
     4     $\text{RuleOut}(e, l_1, l_2)$ ;  
     5    Communicate new *winners*, *p-winners*, *losers* and *s-losers*;  
     6    **if**  $e$  is p-winner  $\wedge$   $e$  is not loser **then** declare  $e$  winner **else** declare  $e$  loser;  
     7    **if**  $e$  is s-looser  $\wedge$   $e$  is not winner **then** declare  $e$  loser;  
     8    Set  $e.\text{lot}$  to  $+\infty$  for *winners* and to  $-\infty$  for *losers*;  
     9     $\text{length} := 2\text{range} - 1$ ;  $\text{range} := 2 \lfloor \log_2 \text{range} \rfloor + 1$ ;  
     **until** ( $R$  contains only elements that are winners or losers)  $\vee$  ( $\text{length} < k$ );  
     **return** Set  $S$  of winners.

---

**3.2 A Randomized Algorithm with Better Performance**

Now we will describe a randomized algorithm for which we will have a better performance than for the deterministic one, as shown in Section 4. It uses the technique of *independent sets*, as described in Jájá (1992). An *independent set* is a subset  $I$  of the list-items such that no two items in  $I$  are neighbors in the list. In fact such a set  $I$  only contains *internal items* i.e. items that are not a head or a tail of one of the sublists. These items in  $I$  are ‘shortcut’ by the algorithm: they inform their left and right neighbors about each other such that they can point to each other directly. The advantage of this technique compared to Algorithm 2 is that the construction of the set that goes into recursion requires only one communication round in each recursive call. To limit the number of supersteps, the depth of the recursion has to be relatively small and this can be ensured if the size of the independent set is sufficiently large in each recursive call. Algorithm 5 solves the LRP with this technique in the CGM model.

It is easy to see that Algorithm 5 is correct. The following is also easy to see with an argument over the convergence of  $\sum_i \varepsilon^i$ , for any  $0 < \varepsilon < 1$ .

**Algorithm 5:** IndRanking(L) List Ranking by Independent Sets

---

**Input:** Family of doubly linked lists  $L$  (linked via  $l[v]$  and  $r[v]$ ) and for each item  $v$  a distance value  $dist[v]$  to its right neighbor  $r[v]$ .

**Output:** For each item  $v$  the end of its list  $t[v]$  and the distance  $d[v]$  between  $v$  and  $t[v]$ .

**if**  $L$  is small **then** send  $L$  to processor 1 and solve the problem sequentially;  
**else**

independent set	Let $I$ be an independent set in $L$ with only internal items and $D = L \setminus I$ ;
→ $D$	<b>foreach</b> $i \in I$ <b>do</b> Send $l[i]$ to $r[i]$ ;
→ $D$	<b>foreach</b> $i \in I$ <b>do</b> Send $r[i]$ and $dist[i]$ to $l[i]$ ;
$I \rightarrow$	<b>foreach</b> $v \in D$ with $l[v] \in I$ <b>do</b>
	Let $nl[v]$ be the value received from $l[v]$ ;
	Set $ol[v] = l[v]$ and $l[v] = nl[v]$ ;
$I \rightarrow$	<b>foreach</b> $v \in D$ with $r[v] \in I$ <b>do</b>
	Let $nr[v]$ and $nd[v]$ be the values received from $r[v]$ ;
	Set $r[v] = nr$ and $dist[v] = dist[v] + nd[v]$ ;
recurse	$IndRanking(D)$ ;
→ $I$	<b>foreach</b> $v \in D$ with $ol[v] \in I$ <b>do</b> Send $t[v]$ and $d[v]$ to $ol[v]$ ;
$D \rightarrow$	<b>foreach</b> $i \in I$ <b>do</b>
	Let $nt[i]$ and $nd[i]$ be the values received from $r[i]$ ;
	Set $t[i] = nt[i]$ and $d[i] = dist[i] + nd[i]$ ;

---

**Lemma 1.** Suppose there is an  $0 < \varepsilon < 1$  for which we ensure for the choices of  $I$  in “independent set” such that  $|I| \geq \varepsilon|L|$ . Then the recursion depth and number of supersteps of Algorithm 5 is in  $O(\log_{1/(1-\varepsilon)} |L|)$  and the total communication and work is in  $O(|L|)$ .

Note that in each recursion round each element of the treated list communicates a constant number of times (at most two times). The values for *small* can be parametrized. If, for instance, we choose *small* equal to  $\frac{n}{p}$ , then the depth of the recursion will be in  $O(\log_{1/(1-\varepsilon)} p)$ , and Algorithm 5 will require  $O(\log_{1/(1-\varepsilon)} p)$  supersteps. Also the total bound on the work depends by a factor of  $1/(1-\varepsilon)$  from  $\varepsilon$ . The communication on the other hand does not depend on  $\varepsilon$ . Every list item is member of the independent set at most once. So the communication that is issued can be directly charged to the corresponding elements of  $I$ . We think that this is an important feature that in fact keeps the communication costs of any implementation quite low.

**Lemma 2.** Suppose every item  $v$  in list  $L$  has value  $A[v]$  that is randomly chosen in the interval  $[1, K]$ , for some value  $K$  that is large enough. Let  $I$  the set of items that have strictly smaller values than their right and left neighbors. Then  $I$  is an independent set of  $L$  and with probability approaching 1 we have that  $|I| \geq \frac{1}{4}|L|$ .

For the implementation side of Algorithm 5 we have to be careful not to spend too much time for (1) initializing the recursion, or (2) choosing (pseudo) random numbers. In fact, we ensure (1) by an array that always holds the active elements, i.e. those elements that were not found in sets  $I$  in recursion levels above. By that we do not have to copy the list values themselves and the recursion does not create any additional communication overhead. For (2), we ensure at the beginning that the list is distributed randomly on the processors. Then every item  $v$  basically uses its own (storage) number as value  $A[v]$ . To ensure that these values are still sufficiently independent in lower levels of recursion we choose for each such level  $R$  a different large number  $N_R$  and set  $A[v] = N_R \cdot v \bmod K$ .

## 4 Implementation

Our main tests for the two algorithms took place on a<sup>1</sup>. It consists of 12 *PentiumPro* 200 PC with 64 MB memory each. The PC are interconnected by a *Myrinet*<sup>2</sup> network of 1.28 Gb/s. The implementation of the algorithm was done –as we think– quite carefully in C++ and based on MPI, one well-known library for message passing between processes. The cluster is equipped with the Linux OS, the g++ compiler from the GNU project and the MPI-BIP implementation (that is an implementation of MPI over Myrinet). The use of C++ allowed us to actually do the implementation on different levels of abstraction: (1) one that interfaces our code to one of the message passing libraries, (2) one that implements the CGM model itself, and (3) the last that implements the specific algorithms for the LRP.

One of our intentions for this hierarchical design is to replace message passing in (1) by shared memory later on. This is out of the scope of the study here, but this shows our wish to have portable code. This later goal seems to be well achieved, since we have been able to run the code on a large variety of architectures: different PC clusters, SUN workstations, a SGI *Origin 2000* and a Cray *T3E*. There, the general outlook of the curves looks similar, certainly that the constant factors are dependent on the architecture, see Guérin Lassous and Gustedt (1999).

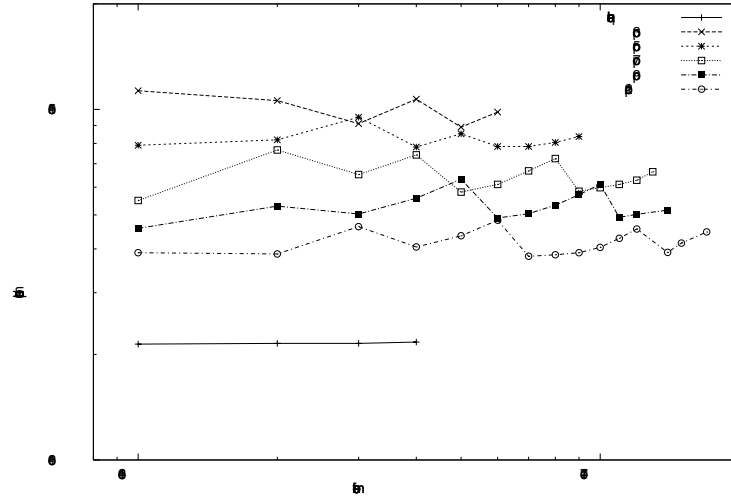
First, we will present the execution times obtained for the two algorithms. Then, we will show in the following that here the CGM model allows a relatively good prediction of what we may expect. Finally, we will show that our algorithms are also good candidates for handling very large lists.

### 4.1 Execution Time

Figure 1 gives the execution times *per element* in function of the list size for Algorithm 2, whereas Figure 2 is for Algorithm 5. To cover the different orders of magnitude better, both axis are given on a *logarithmic* scale. The lists were generated randomly with the use of random permutations. For each list size, the

<sup>1</sup> <http://www.ens-lyon.fr/LHPC/ANGLAIS/popc.html>

<sup>2</sup> <http://www.myri.com/>



**Fig. 1.** Execution times per element for Algorithm 2

program was run (at least) 10 times and the result is the average of these results. For a fixed list size, very small variations in time could be noted.

$p$  varies from 2 to 12 for Algorithm 2 and from 4 to 12 for Algorithm 5. Algorithm 5 is more greedy in memory, therefore the memory of the processors is saturated when we use 2 or 3 PC.

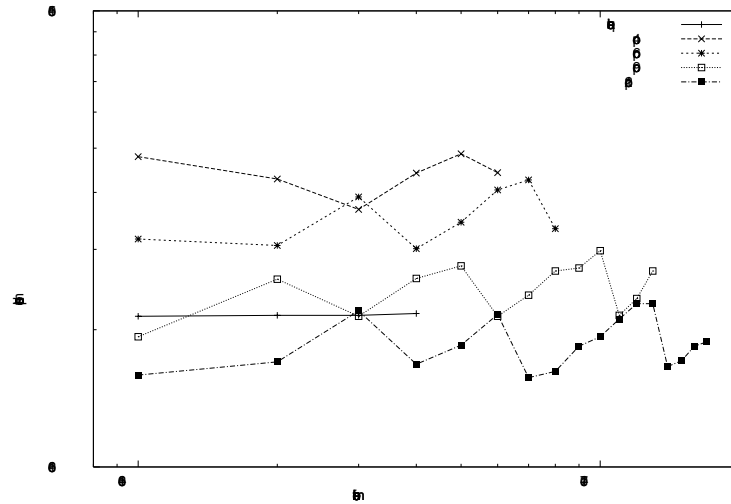
All the curves stop before the memory saturation of the processors. We start the measures for lists with 1 million elements, because for smaller size, the sequential algorithm performs so well that using more processors is not very useful.

Algorithm 2 is always slower than the sequential one. Nevertheless, the parallel execution time decreases with the number of used PC. One might expect that the parallel algorithm becomes faster than the sequential one with the use of more PC (as it is the case with the T3E), but no cluster of more than 12 PC was at our disposal. For Algorithm 5, from 9 PC the parallel algorithm becomes faster than the sequential one. The parallel execution time decreases also with the number of used PC. The absolute speedups are nevertheless small since for 12 PC for instance the obtained speedup is equal to 1.3.

If we compare the two algorithms, we can note that:

1. The use of a larger amount of processors in all cases lead to an improvement on the real execution times of the parallel program, which proves the scalability of our approach,
2. Algorithm 5 is more efficient than Algorithm 2. This easily can be explained by the fact that Algorithm 5 requires less communication rounds and smaller workload and communication costs. We also noted that for Algorithm 5, the size of  $I$  is about one third of  $L$  (compared to the theoretical  $\frac{1}{4}$ ). If we are not able to explain so far, it results that Algorithm 5 works better than expected,





**Fig. 2.** Execution times per element for Algorithm 5

3. Algorithm 5 is greedier in memory than Algorithm 2, therefore we can not use this algorithm with a cluster having less than 4 PC, and
4. Portable code does not lead to effective speedups with a dozen processors.

#### 4.2 Verification of the Complexity

A positive fact that we can deduce from the plots given above is that the execution time for a fixed amount of processors  $p$  shows a linear behavior as expected (whatever the number of used processors may be). One might get the impression from Figure 2, that Algorithm 5 deviates a bit more from linearity in  $n$  (the list size) than Algorithm 2. But this is only a scaling effect: the variation between the values for a fixed  $p$  and  $n$  varying is very small (less than  $1\mu s$ ).

For increasing amount of data and fixed  $p$  the number of supersteps remains constant. As a consequence, the total number of messages is constant, too. So do the costs for initiating messages, which in turn correspond to the offsets of the curves of the total running times. On the other hand, the size of messages varies. But from Figures 1 and 2, we see that the time for communicating data is also linear in  $n$ . Therefore, we can say that, for this problem (that leads to quite sophisticated parallel/distributed algorithms), the local computations and the number of communication rounds are good parameters to predict the qualitative behavior. Nevertheless, they are not sufficient to be able to predict the constants of proportionality and to know the algorithms that will give efficient results or not (as noticed for Algorithm 2).

If moreover, we take the overall workload and communication into account, we see that Algorithm 5 having a workload closer to the sequential one, leads to more efficient results.

### 4.3 Taking Memory Saturation into Account

This picture brightens if we take the well known effects of memory saturation into account. In Figure 1 and Figure 2, all the curves stop before the swapping effects on PC. Due to these effects the sequential algorithm changes its behavior drastically when run with more than 4 million elements. For 5 millions elements, the execution is a little bit bigger than 3000 seconds (which is not far from one hour), whereas Algorithm 2 can solve the problem in 21.8 seconds with 12 PC and Algorithm 5 does it in 9.24 seconds with 12 PC. We see also that to handle lists with 18 millions elements with Algorithm 2 we need 71 seconds and with 17 millions elements with Algorithm 5 18 seconds. Therefore, our algorithms perform well on huge lists.

## References

- [Caceres et al., 1997] Caceres, E., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A., Santoro, N., and Song, S. W. (1997). Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In Degano, P., Gorrieri, R., and Marchetti-Spaccamela, A., editors, *Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Comp. Sci.*, pages 390–400. Springer-Verlag. Proceedings of the 24th International Colloquium ICALP'97.
- [Cole and Vishkin, 1989] Cole, R. and Vishkin, U. (1989). Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):128–142.
- [Dehne et al., 1996] Dehne, F., Fabri, A., and Rau-Chaplin, A. (1996). Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400.
- [Dehne and Song, 1996] Dehne, F. and Song, S. W. (1996). Randomized parallel list ranking for distributed memory multiprocessors. In Jaffar, J. and Yap, R. H. C., editors, *Concurrency and Parallelism, Programming, Networking, and Security*, volume 1179 of *Lecture Notes in Comp. Sci.*, pages 1–10. Springer-Verlag. Proceedings of the Asian Computer Science Conference (ASIAN '96).
- [Guérin Lassous et al., 2000] Guérin Lassous, I., Gustedt, J., and Morvan, M. (2000). Feasibility, portability, predictability and efficiency: Four ambitious goals for the design and implementation of parallel coarse grained graph algorithms. Technical Report 3885, INRIA.
- [Guérin Lassous and Gustedt, 1999] Guérin Lassous, I. and Gustedt, J. (1999). List ranking on a coarse grained multiprocessor. Technical Report RR-3640, INRIA.
- [Guérin Lassous and Gustedt, 2000] Guérin Lassous, I. and Gustedt, J. (2000). List ranking on pc clusters. Technical Report RR-3869, INRIA Lorraine. <http://www.inria.fr/RRRT/publications-fra.html>.
- [Jájá, 1992] Jájá, J. (1992). *An Introduction to Parallel Algorithms*. Addison Wesley.
- [Karp and Ramachandran, 1990] Karp, R. M. and Ramachandran, V. (1990). Parallel Algorithms for Shared-Memory Machines. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume A, Algorithms and Complexity, pages 869–941. Elsevier Science Publishers B.V., Amsterdam.
- [Reid-Miller, 1994] Reid-Miller, M. (1994). List ranking and list scan on the Cray C-90. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 104–113.
- [Sibeyn et al., 1999] Sibeyn, J. F., Guillaume, F., and Seidel, T. (1999). Practical Parallel List Ranking. *Journal of Parallel and Distributed Computing*, 56:156–180.

- [Sibeyn, 1997] Sibeyn, J. (1997). Better trade-offs for parallel list ranking. In *Proc. of 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 221–230.
- [Valiant, 1990] Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.
- [Wyllie, 1979] Wyllie, J. C. (1979). *The Complexity of Parallel Computations*. PhD thesis, Computer Science Department, Cornell University.

# Parallelizing Local Search for CNF Satisfiability Using Vectorization and PVM

Kazuo Iwama, Daisuke Kawai, Shuichi Miyazaki,  
Yasuo Okabe, and Jun Umemoto

School of Informatics, Kyoto University, Kyoto 606-8501, Japan  
{iwama,kawai,shuichi,okabe,umemoto}@kuis.kyoto-u.ac.jp

**Abstract.** The purpose of this paper is to speed up the local search algorithm for the CNF Satisfiability problem. Our basic strategy is to run some  $10^5$  independent search paths simultaneously using PVM on a vector supercomputer VPP800, which consists of 40 vector processors. Using the above parallelization and vectorization together with some improvement of data structure, we obtained 600-times speedup in terms of the number of flips the local search can make per second compared to the original GSAT by Selman and Kautz. We run our parallel GSAT for benchmark instances and compared the running time with those of existing SAT programs. We could observe an apparent benefit of parallelization: Especially, we were able to solve two instances that have never been solved before this paper. We also tested parallel local search for the SAT encoding of the class scheduling problem. Again we were able to get almost the best answer in reasonable time.

## 1 Introduction

Local search is probably the most popular approach to combinatorial optimization problems. It works quite well for almost all different kinds of problems including CNF Satisfiability (SAT) discussed in this paper. From an algorithmic point of view, people's interests have been mainly focused on how to manage local minima, i.e., how to get out of them, efficiently. In the case of SAT, local search approaches were first proposed by [27,11], and subsequently a lot of heuristics for escaping from local minima, such as simple restarting, random walks [26], simulated annealing [29], tabu search and weighting [19,25], have been presented.

Thanks to these efforts, it is said that the size of instances which can be solved in practical time has been increased from some 30–50 variables ten years ago to even  $10^5$  variables recently. One drawback of this approach is, however, that it is very hard to “prove” its performance. Excepting for a very few theoretical analyses [20,16,23], most authors claim the performance of their programs by experiments, which involve intrinsic difficulties like how to select benchmark instances. In this paper, we do not discuss this kind of unguaranteed speedups but we discuss a guaranteed speedup, or what we call, a speedup by implementation. In the case of local search of SAT, the most natural attempt in this category is to increase the number  $N$  of searches or cell-moves per second. This applies to all sorts of algorithms based on different heuristics and apparently gives us a guaranteed speedup due to  $N$ .

For this purpose, it is obviously important to select a best available computing facility. In our case, we selected a vector supercomputer VPP800 which appears to be one of the most powerful computers in our university [6]. VPP800 consists of 40 processors each of which is a so-called vector processor. According to the spec, some 50-times speedup is possible by vectorization in each processor and hence some 2000-times speedup is expected in total. The nature of local search fits parallel computation, since we can execute many search paths in parallel for which little communication is needed. Thus parallelization of local search might be an easy task. However, its vectorization is much more nontrivial. Actually, there are very few examples [13] of vectorizing combinatorial optimization problems. A main contribution of this paper is to develop a “vectorized GSAT.” For several reasons, we are not able to get the ideal 50-times speedup but are able to get some 5-times speedup using a careful implementation. Also we get 40-times speedup by parallelization using Parallel Virtual Machine [8] and 3-times speedup by some data structure. In total, we get 600-times speedup compared to the original GSAT by Selman and Kautz [24].

Due to experiments using the benchmark set for the Second DIMACS Implementation Challenge, Cliques, Coloring and Satisfiability [14], the overall performance of our new approach is generally as we expected. Especially, we were able to solve two instances which were never solved before.

It should be noted that our method of vectorization needs a lot of memory space, which is apparently weak for instances of large size. For example, a CNF formula which encodes a class-scheduling [18] (900 variables and 236,539 clauses) is too large for our new program. (It is possible to run the instance if we decrease the vector-length, which however kills the merit of vectorization.) For this instance, we selected another computation system, namely, a cluster of 70 workstations. By using only PVM, we get a nice result in reasonable computation time. Thus we can conclude that there is an apparent benefit of parallelization for local search of CNF Satisfiability. We believe that there is a similar tendency for local search in general, which will be a nice further research.

## 2 Local Search Algorithms for CNF Satisfiability

A *literal* is a logic variable  $x$  or its negation  $\bar{x}$ . A *clause* is a sum of literals and a *CNF formula* is a product of clauses. A truth assignment is a mapping from the set of variables into  $\{0, 1\}$ . If there is a truth assignment that makes all the clauses of  $f$  true, then we say that  $f$  is *satisfiable*, and otherwise, we say that  $f$  is *unsatisfiable*. The *satisfiability problem* (*SAT* for short) asks if  $f$  is satisfiable or not.

The first local search algorithm for SAT was proposed in early 1990s [27,11], and has received much attention because of its overwhelming performance [19,25,26,2,3,28]. The basic idea of the local search is the following: Let  $A$  be a truth assignment and  $A_i$  be the truth assignment obtained from  $A$  by flipping the value of  $x_i$ . (In this case we say that  $A$  and  $A_i$  are *neighbors* to each other. Note that there are  $n$  neighbors for each assignment if there are  $n$  variables.) Then the *gain* of the variable  $x_i$  (at  $A$ ) is defined to be the number of clauses satisfied by  $A_i$  minus the number of clauses satisfied by  $A$ . The local search first selects a starting assignment randomly, and then, at each step, it flips one of the variables having a maximum gain if the gain is positive. If it reaches a local

minimum, i.e., an assignment such that all variables have zero or negative gains, the algorithm will be stuck. There are several heuristics for “escaping” from a local minimum other than simply restarting the search:

**GSAT** [27,24]: When this algorithm reaches a local minimum, it keeps moving by flipping a variable whose gain is maximum even if it is zero or negative.

**Weighting** [19,25]: When this algorithm reaches a local minimum, it gives (appropriate) weights to clauses so that the current assignment will no longer be a local minimum.

**Random Walk** [26]: At each step (not necessarily at a local minimum), this algorithm selects one of the following strategies with equal probability: (i) execute GSAT, (ii) flip one variable randomly selected from unsatisfied clauses.

Because of its simplicity, we mainly discuss GSAT, especially, the one developed by Selman and Kautz, whose source code is open at [24]. We need two parameters to specify stopping condition of GSAT, MAXFLIPS and MAXTRIES. MAXFLIPS determines the number of flips executed from one starting assignment. We call this series of movements a *try*. MAXTRIES determines the number of tries GSAT executes. Therefore, GSAT executes  $\text{MAXTRIES} \times \text{MAXFLIPS}$  flips in total. As we have seen before, GSAT moves to a neighbor by flipping one of the variables having maximum gain. However, we adopt an option supported by [24]: The algorithm chooses a variable randomly among ones that have positive gains. (If there is no variable of positive gain, then choose one among zero-gain variables. If there are only negative-gain variables, then choose one randomly among all variables.) It is reported that this option does not lose efficiency, namely, the success rate of finding solutions does not change much [10].

### 3 Speedups by Improving Data Structure

At each step of the original GSAT, it calculates the number of unsatisfied clauses for each neighbor. There are  $n$  neighbors, and for each neighbor obtained from the current assignment by flipping  $x_i$ , it suffices to examine clauses that contain  $x_i$ . Hence, roughly speaking, the computing time for one step is  $O(nl)$ , where  $l$  is the maximum number of appearances of variables. Selman and Kautz [24] exploited so-called a *gain table* and *gain buckets* to reduce the time needed for this one step.

A gain table maintains a gain of each variable. A gain bucket is a set of variables: The *positive bucket* is the set of variables whose gains are positive, the *zero bucket* is the set of variables whose gains are zero, and the *negative bucket* is the set of variables whose gains are negative. Our task is to choose one variable from the positive bucket, update the gain table and update each bucket. For example, consider a clause  $C = (x_1 + \bar{x}_2 + x_3)$ . Suppose that  $x_1 = 1$ ,  $x_2 = 1$  and  $x_3 = 0$  under the current assignment, say  $A$ , and that the value of  $x_1$  will be flipped (let the resulting new assignment be  $A'$ ). At the assignment  $A$ , the clause  $C$  does not contribute to gains of  $x_2$  and  $x_3$  because flipping  $x_2$  or  $x_3$  does not change the satisfaction of  $C$  at all. However,  $C$  contributes a negative ( $= -1$ ) effect to the gain of  $x_1$  because flipping  $x_1$  from 1 to 0 makes the clause unsatisfied. At  $A'$ ,  $C$  contributes  $+1$  to each of  $x_1$ ,  $x_2$  and  $x_3$  because each flip changes  $C$  from unsatisfied to satisfied. Hence, by flipping  $x_1$ , the gains of  $x_1$ ,

$x_2$  and  $x_3$  increase by 2, 1 and 1, respectively, by this clause  $C$ . (Of course, we need to check all the clauses that contain  $x_1$ .)

Generally speaking, choosing a variable takes  $O(n)$  time. Updating a gain table and gain buckets takes  $O(l)$  time: There are at most  $l$  clauses that contain the flipped variable  $x$ . Then there are at most  $2l$  variables other than  $x$  whose gain changes (if each clause contains at most three literals). Only these variables may change their gains. Thus the computing time for each step reduces from  $O(nl)$  to  $O(n + l)$  by using the bucket structure.

We improve this three-buckets structure: Recall that we need the following three procedures at each step; (i) choose a positive-gain variable randomly and flip it, (ii) update a gain table, and (iii) update each gain bucket. In this section, we concentrate on reducing time for (iii) by improving the data structure.

### 3.1 Removing the Negative Bucket

As mentioned above, the advantage of using gain buckets is to facilitate selecting a variable to be flipped. Now let us suppose that we are to flip a negative-gain variable. This means that there is no variable of positive-gain nor zero-gain, or that all the variables have negative-gain, namely, we are merely choosing one variable from all variables. Hence the negative bucket can be omitted.

By removing the negative bucket, we can reduce the cost of adding variables to the negative bucket and deleting variables from the negative bucket. We can expect speedup by this method because, as the search proceeds, there are very few movements of variables between positive and zero buckets. In most cases, variables move between the zero bucket and the negative bucket.

### 3.2 Using Indices

Consider, for example, the gain of the variable  $x_2$  changed from positive to zero by some flip. Then we have to move  $x_2$  from the positive bucket to the zero bucket. A simple method of realizing each bucket is to use a linear list, namely,  $\text{pb}[i]$  represents the  $i$ -th item (variable) in the bucket. Searching the list takes linear time, i.e., a time proportional to the size of the positive bucket in the worst case. Instead, we prepare an index for each variable which indicates the position of the variable in the positive buckets so that random access is possible.

### 3.3 Experiments

We compared the efficiency of GSAT without buckets (i.e., the original GSAT), three-buckets GSAT (i.e., the one by Selman and Kautz), two-buckets GSAT and two-buckets GSAT using indices. All these programs were run on SUN workstation UltraSPARC-IIi (300 MHz). Table 1 shows the number of flips each program made per second. We used randomly generated satisfiable instances where each clause contains exactly three variables [30]. Variable/clause ratio was set to 4.3, which is known to produce hard random 3SAT instances [5,17]. We used 500-variable, 1000-variable and 2000-variable instances. For each case, we run each program for 10 different instances and took the average value of the number of flips. MAXFLIPS was set to 10 times the number of variables and MAXTRIES was set to 10. Our two buckets and indices method gave almost 3-times speedup compared to Selman and Kautz's GSAT.

**Table 1.** The number of flips per second

# variables	500	1000	2000
Without buckets	74,405	44,964	23,481
3 buckets	103,999	101,437	95,696
2 buckets	265,957	226,244	169,492
2 buckets and indices	280,899	276,243	222,469

## 4 Speedups by Vectorization

Here is our main theme of this paper, vectorization. Let us recall the computation of GSAT. For each try, GSAT selects a starting assignment randomly, and moves from assignments to assignments by flipping the value of variables. At each step, it selects the variable being flipped, and updates the gain table and gain buckets. Thus the computation at each step completely depends on the variable selected in the previous steps, namely, we cannot start the computation of step  $i$  before the computation of step  $i - 1$  is completed. Hence, during one try, the parts of the computation which can be vectorized are very limited: We can vectorize only computations of selecting a starting assignment and calculating gains at the initial step, by which we cannot expect satisfactory speedups.

However, two different tries are completely independent because they start from their own starting assignments independently, and execute movements depending only on those starting assignments (and some randomness). We adopt this type of parallelization, i.e., running multiple tries simultaneously.

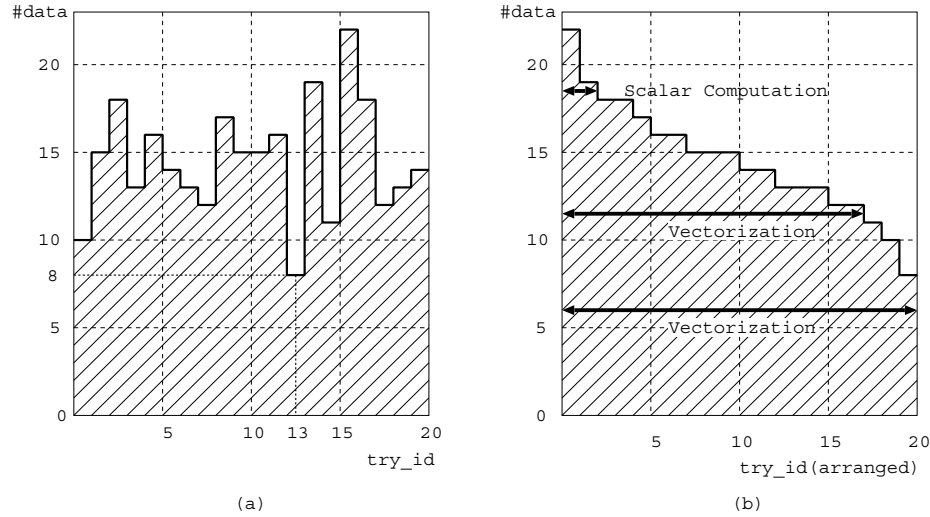
### 4.1 Vectorization

For the above purpose, our first step is to introduce an index `try_id` for all variables used in GSAT program. `try_id` denotes the try number. We run 2048 tries in parallel, so `try_id` takes the values 1 through 2048. We rewrite GSAT program so that the loop for `try_id` becomes the most inner loop and vectorize this loop.

There is one problem when vectorizing. Suppose that, for simplicity, 20 tries are parallelized. Then, each try has to manage a different size of data. Consider for example that GSAT refers the indices of variables whose gains changed by flipping one variable. Fig. 1 (a) illustrates an example of the number of such data for each try. A vector machine executes read operations row by row from the bottom in Fig. 1 (a). Then, the program fails when executing the 9th row because the 13th try has only 8 data and the vector data are cut here. To avoid this problem, we prepare an “if” statement which judges whether data actually exist, and vectorize whole the row by masked vector operation, which can be executed by one instruction. Simply speaking, we prepare dummy to fill such blanks.

Now the above fail does not occur, but the next problem is the efficiency. Again, see Fig. 1 (a). When the program executes the first through eighth rows, it executes 20 operations at a time, and hence we can enjoy the benefits of long vectors. However, when it executes 20th row, it executes only one operation and we can have no advantage of vectorization around there, or it may be even worse than simply executing only that data. Our strategy for resolving this problem is





**Fig. 1.** An example of the number of data for each try

shown in Fig. 1 (b). We gather only the portions where data actually exist, and vectorize only those portions. Furthermore, when the number of data becomes small, for example, two or three around 17th row of Fig. 1 (b), we switch to the scalar computation since vectorization loses efficiency if the length of the vector is too short.

## 4.2 Experiments

We ran the following six programs to see the effect of vectorization and gathering, and to see what type of gathering is most effective. All the programs below are based on the same algorithm described in the previous section, which uses two buckets and indices:

- 1: We do not use vectorization.
- 2: We use vectorization but do not use gathering.
- 3: We use gathering at every row.
- 4: We use gathering at every other row.
- 5: We do not use gathering at the first three rows (because all tries are likely to have data at the very first rows), and after that, we use gathering at every row.
- 6: We compute the minimum number  $\min$  of data among all tries (8 in the example of Fig. 1 (b)) in advance. We do not use gathering at the first  $\min$  rows, and after that, use gathering at every row.

Programs 2 through 6 use vectorization. Among them, programs 3 through 6 use gathering. Programs 3 through 6 switch to the scalar computation when the number of data becomes very small.

Table 2 compares the number of flips each program made per second. All six programs were run on the parallel vector supercomputer Fujitsu VPP800 at Kyoto University Data Processing Center (1 GOPS (Giga operations per second))

**Table 2.** The number of flips per second for several ways of gathering

# variables		100	200
1		287,799	312,898
vector	2	840,699	858,256
	3	1,558,935	1,691,241
	4	1,404,014	1,530,819
	5	1,494,083	1,612,150
	6	1,553,924	1,707,838

and 8 GFLOPS (Giga floating-point Operations per second) per CPU) [6]. Vectorized programs (i.e., programs 2 through 6) execute 2048 tries simultaneously. Again, we used randomly generated satisfiable instances whose variable/clause ratio is 4.3 [30]. This time, we used instances of 100 variables and 200 variables. For each case, we run programs for 100 instances and took their average. MAXFLIPS was set to 10 times the number of variables, and MAXTRIES was set to 2048 for the program 1, and 1 for programs 2 through 6 (so that the total numbers of flips are equal for all programs). Success ratio, i.e., the ratio of the number of instances the program found a satisfying assignment to the number of all instances are almost the same for all six programs; Almost 100% for 100-variable instances and about 50% for 200-variable instances.

As one can see, a simple vectorization gives us approximately 2.5-times speedup, and gathering gives us further speedup of twice. However, the way of gathering does not make a big difference. One may think that 5-times speedup is poor since 2048 tries are executed parallelly. This is mainly because of the vector indirect addressing which is inherent in vectorization of combinatorial algorithms. In the next section, we do further parallelization using Parallel Virtual Machine (PVM).

## 5 Speedups by PVM

Our next (and final) step is to use multiple processors to run our currently best program (i.e., the one using two buckets with indices, and vectorization with gathering). For this purpose, our selection is Parallel Virtual Machine (PVM) [8]. PVM is a popular software by which we can connect a great number of processors easily and use them as if they form a parallel computer.

The parallelization method we use here is simple: One processor serves as a master and all other processors serve as slaves. Each slave processor runs a try (actually vectorized 2048 tries), and when it is completed, the slave processor informs the completion to the master processor. The master processor counts the number of completed tries, and when it reaches MAXTRIES, the program terminates. Or, some slave processor finds a solution, the program terminates.

Usually, the bottleneck of the distributed computation is communication between processors. However, the above method requires almost no communication, by which we can expect almost  $t$ -times speedup by using  $t$  processors.

**Table 3.** The number of flips when 1 CPU and 40 CPUs are used

# variables	100	200	500
1CPU	1,677,181	1,657,016	1,726,874
40CPUs	63,269,478	63,825,510	67,817,472
Ratio	37.72	38.52	39.28

### 5.1 Experiments

For experiments, we used vector supercomputer VPP800 [6] again. To examine the effect of parallelization, we compared the number of flips per second by 40 CPUs and by 1 CPU (Table 3). The GSAT program we adopted here is the vectorized one using gathering everytime, and two buckets with indices. We conducted experiments for random formulas of 100 variables, 200 variables and 500 variables. For each case, we run the above GSAT for 5 instances and took the average. This time, we used unsatisfiable instances [1] because we needed to run the program for a sufficiently long time to take the accurate data. For each case, we obtained almost 40-times speedup as we have expected.

## 6 DIMACS Benchmarks

Using several heuristics as we have shown so far, 600-times speedup has been achieved compared to GSAT by Selman and Kautz in terms of the number of flips. To see the real benefits of our program, we run our GSAT for benchmark instances of the 2nd DIMACS Implementation Challenge, Cliques, Coloring and Satisfiability [14]. We used only satisfiable instances. Each row of Table 4 shows the name, the number of variables and the number of clauses of an instance, and time (Sec.) to find a satisfying assignment of that instance by each program. Figures under the decimal point are rounded down. Especially, 0 means that the answer was obtained in less than a second. Our GSAT was run on VPP800 as before and Selman and Kautz's GSAT was run on UltraSPARC-IIi (300 MHz). We run those two GSAT programs for 6 hours for each instance. In Table 4, the "N.A." sign (for "No Answer") means that it was not possible to find a solution within 6 hours. For other programs, we just transcribed the results given in each paper in [14]. The blank entry means that the program was not run for that instance, and "N.A." has the same meaning as above. (The running time was not specified in [14].) Table 5 shows the authors and the name of each program.

To examine the difference of machine performance, DIMACS provides five test programs. The rightmost column of Table 5 shows the average ratio of running time of test programs on each machine over the runtime on VPP800 scalar computation. (For example, the machine used for the program No. 6 [29] is 4.63 times slower than VPP 800.)

Selman and Kautz's GSAT and programs No.2 and 6 are based on the local search algorithm. Programs 1, 3, 4 and 7 are based on backtracking. One can see that **g** instances are easier for the local search, while **par** instances are easier for backtracking. Our parallel GSAT is not only faster than existing local search programs but also comparable to backtracking even for instances which are much easier for backtracking, e.g., **par** instances.

Finally, we parallelized Random Walk introduced in Sec. 2, using the same technique, and run for some DIMACS instances for which our parallel GSAT

**Table 4.** Results for DIMACS benchmarks (Each entry is the running time (Sec.) for solving the corresponding instance by the corresponding program. Figures under the decimal point are rounded down. The “N.A.” sign means that the program failed. The blank entry means that the program was not run for that instance. See Sec. 6 for details.)

Instance Name	# variables	# clauses	Runtime (Sec.)									
			Our GSAT	Selman's GSAT	Other SAT programs [14]							
					1	2	3	4	5	6	7	
aim-100-2.0-yes1-1	100	200	1	227	0	17	135	2	398	N.A.	1	
aim-100-2.0-yes1-2	100	200	2	96	0	29	2	5	239	13,929	1	
aim-100-2.0-yes1-3	100	200	1	49	0	13	17	1	63	22,500	1	
aim-100-2.0-yes1-4	100	200	2	226	0	15	0	0	1,456	N.A.	0	
f400.cnf	400	1,700	3	48	2,844	34	5,727	210,741	60	10,870		
f800.cnf	800	3,400	182	N.A.		1,326				27,000		
f1600.cnf	1,600	6,800	*509	N.A.						N.A.		
f3200.cnf	3,200	13,600	*19,840	N.A.						N.A.		
g125.17.cnf	2,125	66,272	261	N.A.		103,310			N.A.	453,780	N.A.	
g125.18.cnf	2,250	70,163	17	4		126			N.A.	480	N.A.	
ii32b3.cnf	348	5,734	15	55	2	4	4	1	1	5,400	17	
ii32c3.cnf	279	3,272	8	3	1	3	0	5	1	12,180		
ii32d3.cnf	824	19,478	38	25	973	19	10	3	20	1,200	N.A.	
ii32e3.cnf	330	5,020	11	0	1	3	4	1	3	3,900	3	
par16-2-c.cnf	349	1,392	183	N.A.	23		329	48	1,464	N.A.	145	
par16-4-c.cnf	324	1,292	554	N.A.	6		210	116	1,950	N.A.	145	
ssa7552-159.cnf	1,363	3,032	*30	N.A.	1	82	6	1	1	N.A.	1	
ssa7552-160.cnf	1,391	3,126	*40	N.A.	1	86	6	1	23	175,500	1	

\* Result by Random Walk

**Table 5.** Names and authors of SAT programs and machine performance

No.	Authors	Program name	Performance
1	Dubois, et. al [7]	C-SAT (backtracking)	3.34
2	Hampson & Kibler [12]	Cyclic, Opportunitistic Hill-Climbing	7.43
3	Jaumard, et.al [15]	Davis-Putnum-Loveland	10.82
4	Pretolani [21]	H2R (Davis-Putnum-Loveland) BRR (B-reduction)	18.03
5	Resende & Feo [22]	Greedy	1.84
6	Spears [29]	SASAT (local search with simulated annealing)	4.63
7	Gelder & Tsuji [9]	2cl (branching and limited resolution)	3.14

failed. “\*” mark in Table 4 shows the results by parallel Random Walk. This program solved **f1600.cnf** within 8 minutes, and **f3200.cnf** within 5 and half hours, both of which have never been solved before this paper.

## 7 Real World Instances

Finally, we examine the merit of parallelization for real world problems. Here we use a CNF formula which encodes the class scheduling problem [18]. This problem is to generate a class-schedule table which minimizes the number of broken constraints (which we call a *cost*). Our SAT approach in [18] could find a class-schedule table whose cost is 93 in a few days, while CSP (constraint satisfaction problem) approach found a class-schedule of cost 86 (which is claimed to be optimal) quickly [18,4].

Recall that our vectorized GSAT runs 2048 independent tries on each processor, which means we need prepare 2048 independent data (variables, arrays and so on) in each processor. This needs a lot of memory space if the instance is large. Actually it seems hard to run the class-schedule benchmark (which consists of about 230,000 clauses) using the 2048 parallel version. If we decrease the degree of parallelism, it is possible to run the instance but it loses the merit of long vectors significantly. For this large instance, therefore, we gave up the vectorization and used only PVM for the cluster of 70 workstations. The result is not bad: We were able to get an answer of cost 87 in some two hours of computation.

### 7.1 Class Scheduling Problem

An instance of the class scheduling (CLS) problem consists of the following information: (i) A set  $\Sigma_S$  of students,  $\Sigma_P$  of professors,  $\Sigma_R$  of classrooms,  $\Sigma_T$  of timeslots and  $\Sigma_C$  of courses. (ii) Which courses each student in  $\Sigma_S$  wishes to take, e.g., student  $s$  wants to take courses  $c_1, c_2$  and  $c_3$ . (iii) Which courses each professor in  $\Sigma_P$  teaches, (iv) Which timeslots each professor cannot teach, (v) Which courses each classroom cannot be used for (because of its capacity).

The CLS problem requires to generate a class-schedule. Take a look at the following conditions: (a) The table includes all the courses. (b) At most one course is taught in each timeslot and in each room. (c) If two courses  $c_1$  and  $c_2$  are taught by the same professor,  $c_1$  and  $c_2$  must be assigned to different timeslots. (d) If two courses  $c_1$  and  $c_2$  are selected by the same student,  $c_1$  and  $c_2$  must be assigned to different timeslots. (e) If a course  $c_1$  is taught by the professor who cannot (or does not like to) teach in timeslot  $t_1$ , then  $c_1$  must not be assigned to  $t_1$ . (f) If course  $c$  cannot be taught in room  $r$ ,  $c$  must not be assigned to  $r$ .

Conditions (a) through (c) are mandatory, and conditions (d) through (f) are optional: If  $k$  students wish to take both  $c_1$  and  $c_2$ , then the cost of the constraint “ $c_1$  and  $c_2$  must be assigned to different timeslots” in the condition (d) is  $k$ . All constraints in (e) and (f) has the cost of 1. The cost of the time-schedule is defined to be the sum of the costs of constraints it breaks. Our aim is to generate a class-schedule whose cost is minimum.

### 7.2 Translation Algorithm

Let  $A, B$  and  $C$  be the numbers of the total courses, timeslots and classrooms. Then we use variables  $x_{i,j,k}$  ( $1 \leq i \leq A, 1 \leq j \leq B$ , and  $1 \leq k \leq C$ ). Namely  $x_{i,j,k} = 1$  means that course  $c_i$  is assigned to timeslot  $t_j$  and room  $r_k$ . Now we generate a formula  $f$  from the above information (i)–(v) as follows:

*Step 1.* For each  $i, 1 \leq i \leq A$ , we generate the clause  $(x_{i,1,1} + x_{i,1,2} + \cdots + x_{i,B,C})$ .

This clause becomes false if course  $c_i$  is not taught.

*Step 2.* For each  $i_1, i_2, j, k (i_1 \neq i_2)$ , we generate the clause  $(\overline{x_{i_1,j,k}} + \overline{x_{i_2,j,k}})$ , which becomes false if different courses  $i_1$  and  $i_2$  are taught in the same room at the same time.

*Step 3.* Suppose for example that professor  $p_1$  teaches courses  $c_2, c_4$  and  $c_5$ . Then for each  $j, k_1, k_2 (k_1 \neq k_2)$ , we generate the clauses  $(\overline{x_{2,j,k_1}} + \overline{x_{4,j,k_2}})$   $(\overline{x_{2,j,k_1}} + \overline{x_{5,j,k_2}})$   $(\overline{x_{4,j,k_1}} + \overline{x_{5,j,k_2}})$   $(\overline{x_{2,j,k_1}} + \overline{x_{2,j,k_2}})$   $(\overline{x_{4,j,k_1}} + \overline{x_{4,j,k_2}})$   $(\overline{x_{5,j,k_1}} + \overline{x_{5,j,k_2}})$ . If two courses (including the same one) taught by the same professor  $p_1$  are

assigned to the same timeslot and different rooms, then at least one of those clauses becomes false. We generate such clauses for each of all the professors.

Clauses of Steps 4 through 6 are generated according to conditions (d) through (f) in the similar way. Clauses generated in Steps 1 through 3 are mandatory and our task is to find a truth assignment that satisfies all those clauses and as many clauses in Steps 4 through 6 as possible. We used the same instance as [18] which is based on real data of Computer Science Department of Kyushu University. It includes 13 professors, 60 students, 30 courses, 10 timeslots and 3 rooms. The resulting formula includes 900 variables and some 230,000 clauses, of which about 15,000 clauses are mandatory.

### 7.3 Experiments

Following the argument in [18,4], we gave the weight of 20 to each mandatory clause so that those clauses tend to be satisfied. We used Weighting method (see Sec. 2) using PVM only on seventy SGI O2 (180MHz) which are connected by FastEthernet. As we have mentioned above, we were able to obtain a time-schedule whose cost is 87 within two hours.

## References

1. Asahiro, Y., Iwama, K. and Miyano, E., "Random Generation of Test Instances with Controlled Attributes," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.377–393, 1996.
2. Cha, B. and Iwama, K., "Performance test of local search algorithms using new types of random CNF formulas," *Proc. IJCAI-95*, pp.304–310, 1995.
3. Cha, B. and Iwama, K., "Adding new clauses for faster local search," *Proc. AAAI-96*, pp.332–337, 1996.
4. Cha, B., Iwama, K., Kambayashi, Y. and Miyazaki, S., "Local search algorithms for Partial MAXSAT," *Proc. AAAI-97*, pp.263–268, 1997.
5. Cheeseman, P., Kanefsky, B. and Taylor W.M., "Where the really hard problems are," *Proc. IJCAI-91*, pp.331–337, 1991.
6. Data Processing Center, Kyoto University. Vector Parallel Supercomputer VPP800/63, <http://www.kudpc.kyoto-u.ac.jp/Supercomputer/>
7. Dubois, O., Andre, P., Boufkhad, Y. and Carlier, J. "SAT versus UNSAT," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.415–436, 1996.
8. Geist, A., Berguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V., *Parallel Virtual Machine*, The MIT Press, 1994.
9. Gelder, A.V. and Tsuji, Y.K. "Satisfiability testing with more reasoning and less guessing," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.5590–586, 1996.
10. Gent, I., and Walsh, T., "Unsatisfied variables in local search," *Hybrid Problems, Hybrid Solutions (AISC-95)*, Amsterdam, 1995.
11. Gu, J. "Efficient local search for very large-scale satisfiability problems," *Sigart Bulletin*, Vol.3, No.1, pp.8–12, 1992.

12. Hampson, S. and Kibler, D. "Large plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.437–455, 1996.
13. Hiraishi, H., Hamaguchi, K., Ochi, H. and Yajima, S., "Vectorized Symbolic Model Checking of Computation Tree Logic for Sequential Machine Verification," *Proc. CAV'91 (LNCS 575)*, pp.214–224, 1991.
14. Johnson, D. S. and Trick, M. A. Eds. "Cliques, Coloring, and Satisfiability," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, 1996.
15. Jaumard, B., Stan, M. and Desrosiers, J., "Tabu search and a quadratic relaxation for satisfiability problem," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.457–478, 1996.
16. Koutsoupias, E. and Papadimitriou, C.H., "On the greedy algorithm for satisfiability," *Information Processing Letters*, Vol. 43, pp.53–55, 1992.
17. Mitchell, D., Selman, B. and Levesque, H., "Hard and easy distributions of SAT problems," *Proc. AAAI-92*, pp.459–465, 1992.
18. Miyazaki, S., Iwama, K. and Kambayashi, Y., "Database queries as combinatorial optimization problems," *Proc. CODAS'96*, pp.448–454, 1996.
19. Morris, P., "The breakout method for escaping from local minima," *Proc. AAAI-93*, pp.40–45, 1993.
20. Papadimitriou, C.H., "On selecting a satisfying truth assignment," *Proc. FOCS'91*, pp.163–169, 1991.
21. Pretolani, D., "Efficiency and sability of hypergraph SAT algorithms," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.479–498, 1996.
22. Resende, M.G.C. and Feo, T.A., "A GRASP for satisfiability," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.499–520, 1996.
23. Schoning, U., "A probabilistic algorithm for  $k$ -SAT and constraint satisfaction problems," *Proc. FOCS'99*, pp.410–414, 1999.
24. Selman, B. and Kautz, H., GSAT USER'S GUIDE Version 35, [ftp://ftp.research.att.com/dist/ai/GSAT\\_USERS\\_GUIDE](ftp://ftp.research.att.com/dist/ai/GSAT_USERS_GUIDE), 1993.
25. Selman, B. and Kautz, H., "An empirical study of greedy local search for satisfiability testing," *Proc. AAAI-93*, pp.46–51, 1993.
26. Selman, B. and Kautz, H., "Local search strategies for satisfiability testing," 2nd DIMACS Challenge Workshop, 1993.
27. Selman, B., Levesque, H.J. and Mitchell, D.G., "A new method for solving hard satisfiability problems," *Proc. AAAI-92*, pp.440–446, 1992.
28. Selman, B., Kautz, H. and Cohen, B., "Local Search Strategies for Satisfiability Testing," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.521–531, 1996.
29. Spears, W.M. "Simulated annealing for hard satisfiability problems," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, American Mathematical Society, pp.533–557, 1996.
30. Watanabe Lab, Dept. of Comp. Science, Tokyo Institute of Technology. Random Generation of Unique Solution 3SAT Instances,  
<http://www.is.titech.ac.jp/~watanabe/gensat/a1/index.html>

# Asymptotic Complexity from Experiments? A Case Study for Randomized Algorithms

Peter Sanders<sup>1,\*</sup> and Rudolf Fleischer<sup>2</sup>

<sup>1</sup> Max Planck Insitut für Informatik  
Saarbrücken, Germany  
`sanders@mpi-sb.mpg.de`

<sup>2</sup> Department of Computer Science  
University of Waterloo  
200 University Avenue West  
`rudolf@uwaterloo.ca`

**Abstract.** In the analysis of algorithms we are usually interested in obtaining closed form expressions for their complexity, or at least asymptotic expressions in  $\mathcal{O}(\cdot)$ -notation. Unfortunately, there are fundamental reasons why we cannot obtain such expressions from experiments. This paper explains how we can at least come close to this goal using the scientific method. Besides the traditional role of experiments as a source of preliminary ideas for theoretical analysis, experiments can test falsifiable hypotheses obtained by incomplete theoretical analysis. Asymptotic behavior can also be deduced from stronger hypotheses which have been induced from experiments. As long as a complete mathematical analysis is impossible, well tested hypotheses may have to take their place. Several examples for probabilistic problems are given where the average complexity can be well approximated experimentally so that the support for the hypotheses is quite strong. Randomized Shellsort has performance close to  $\mathcal{O}(n \log n)$ ; random polling dynamic load balancing between  $P$  processors achieves full load sharing in  $\log_2 P + \mathcal{O}(\log \log P)$  steps; randomized writing to  $D$  independent disks using a shared buffer size  $W$  achieves average efficiency at least  $1 - D/(2W)$ .

## 1 Introduction

The complexity analysis of algorithms is one of the core activities of computer scientists in general and in the branch of theoretical computer science known as algorithmics in particular. The ultimate goal would be to find closed form expressions for the runtime or other measures of resource consumption. Since this is often too complicated, we are usually content with asymptotic expressions for the worst case complexity depending on input parameters like the problem size. Even this task can be very difficult so that it is important to use all available tools.

---

\* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).



This paper investigates to what extent experiments can help to find these expressions. It is common practice to plot complexity measures derived from experiments to generate conjectures. But people are rightfully wary about over-interpretation of these results so that the experimental “scaffolding” usually disappears from the publication of the results. Here, it is explained why with some care experiments can often play a stronger role. One way to make the meaning of “some care” more precise is to apply the terminology of the scientific method [15]. The scientific method views science as a cycle between theory and practice. Theory can inductively or (partially) deductively formulate falsifiable hypotheses which can be tested by experiments. The results may then yield new or refined hypotheses. This mechanism is widely accepted in the natural sciences and is often viewed as a key to the success of these disciplines.

Sect. 2 reviews some of the main problems and explains how to partially solve them. Sect. 3 surveys some related work and Sect. 4 gives several concrete examples for randomized algorithms whose expected resource consumption only depends on the input size but which are nontrivial to analyze analytically. Finally, Sect. 5 discusses the role of results found using the scientific method.

## 2 Some Problems with Experiments

*Too Many Inputs:* Perhaps the most fundamental problem with experiments in algorithmics is that we can rarely test all possible inputs even for bounded input size because there are usually exponentially many of them. If we do application oriented research this problem may be mitigated by libraries of test instances which are considered “typical” (e.g. [5]). Here we want to concentrate on experiments as a tool for theory however so that we are interested in cases where it is possible to bound the complexity for all inputs of size  $n$  in time polynomial in  $n$ . For example, there is a large class of *oblivious* algorithms where the execution time only depends on a small number of parameters like the input size, for example, matrix multiplication. Although many oblivious algorithms are easy to analyze directly, experiments can sometimes help. Furthermore, there are algorithmic problems with few inputs. For example, the locality properties of several space filling curves were first found experimentally and then proven analytically. Later it turned out that a class of experiments can be systematically converted into theoretical results valid for arbitrary curve sizes [14].

Experiments are more important for randomized algorithms. Often randomization can convert all instances into average case instances. For example, every sorting algorithm which is efficient on the average can be transformed into an efficient algorithm for worst case inputs by permuting the inputs randomly. In this case, a few hundred experiments with random inputs can give a reliable picture of the expected performance of the algorithm for inputs of the given size. On the other hand, closed form analysis of randomized algorithms can be very difficult. For example, the average case performance of Shellsort is open for a long time now [19]. Also refer to Sect. 4.3.

*Unbounded Input Size:* Another problem with experiments is that we can only test a finite number of input sizes. For example, assume we find that some sorting algorithm needs an average of  $C(n) \leq 3n \log n$  comparisons for  $n < 10^6$  elements. We still cannot claim that  $C(n) \leq 3n \log n$  is a theorem since quadratic behavior might set in for  $n > 42 \cdot 10^6$ . Here, the scientific method partially saves the situation. We can formulate the hypothesis  $C(n) \leq 3n \log n$  which is scientifically sound since it can be falsified by presenting an instance of size  $n$  with  $C(n) > 3n \log n$ . Note that not every sound hypothesis is a good hypothesis. For example, if we would cowardly change the above hypothesis to  $C(n) \leq 100000n \log n$  it would be difficult to falsify it even if it later turns out that the true bound is  $C(n) = n \log n + 0.1n \log^2 n$ . But qualitative issues like accuracy, simplicity, and generality of hypotheses are also an issue in the natural sciences and this does not hinder people to use the scientific method there.

*$\mathcal{O}(\cdot)$ -s Are not Falsifiable:* The next problem is that asymptotic expressions cannot be used directly in formulating a scientific hypothesis since it could never be falsified experimentally. For example, if we claim that a certain sorting algorithm needs at most  $C(n) \leq \mathcal{O}(n \log n)$  comparisons it cannot even be falsified by a set of inputs which clearly shows quadratic behavior since we could always claim that this quadratic development would stop for sufficiently large inputs. This problem can be solved by formulating a hypothesis which is stronger than the asymptotic expression we really have in mind. The hypothesis  $C(n) \leq 3n \log n$  used above is a trivial example. A less trivial example is given in Sect. 4.3.

*Complexity of the Machine Model:* Although the actual execution time of an algorithm is perhaps the most interesting subject of analysis, this measure of resource consumption is often difficult to model by closed form expressions. Caches, virtual memory, memory management, compilers and interference with other processes all influence execution time in a difficult to predict way.<sup>1</sup> At some loss of accuracy, this problem can be solved by counting the number of times a certain set of source code operations is executed which cover all the inner loops of the program. This count suffices to grasp the asymptotic behavior of the code in a machine independent way. For example, for comparison based sorting algorithms it is usually sufficient to count the number of key comparisons.

*Finding Hypotheses:* Except in very simple cases, it is almost impossible to guess an appropriate formula for a worst case upper bound given only measurements; even if the investigated resource consumption only depends on the input size. The measured function may be nonmonotonic while we are only interested in a monotonic upper bound. There are often considerable contributions of lower order terms for small inputs. Experience shows that curve fitting often won't

---

<sup>1</sup> Remember that the above complexity is also an argument *in favour* of doing experiments because the full complexity of the hardware is difficult to model theoretically. We only mention it as a problem in the current context of inducing asymptotic expressions from experiments.

work in particular if we are interested in fine distinctions like logarithmic factors [12]. Again, the scientific method helps to mitigate this problem. Often, we are able to handle a related or simplified version of the system analytically or we can make “heuristic” steps in a derivation of a theoretical bound. Although the result is not a theorem about the target system, it is good enough as a hypothesis about its behavior in the sense of the scientific method. Sect. 4 gives several examples of this powerful approach which so far seems to be underrepresented in algorithmics.

### 3 Related Work

The importance of experiments in algorithm design has recently gained much attention. New workshops (ALENEX, WAE) and journals (ACM J. of Experimental Algorithmics) have been installed and established conferences (e.g., SODA, ESA) explicitly call for experimental work. Using the scientific method as a basis for algorithmics was proposed by Hooker [6]. McGeoch, Precup and Cohen [12] give heuristic algorithms for finding upper bounds on measured function values which are found to be reliable within a factor  $\sqrt{n}$ . They stress that finding more accurate bounds would be futile in general. This is no contradiction to the examples given in Sect. 4 where even  $\log \log n$  terms are discussed because Sect. 4 uses additional problem specific information via the scientific method.

### 4 Examples

Our first example in Sect. 4.1 can be viewed as the traditional role of experiments as a method to generate conjectures on the behavior of algorithms but it has an additional interpretation where the experiment plus theory on a less attractive algorithm yields a useful hypothesis. Sect. 4.2 gives an example where an experiment is used to validate a simplification made in the middle of a derivation. Sections 4.3 and 4.4 touch the difficult question of how to use experiments to learn something about the asymptotic complexity of an algorithm. In addition, Sect. 4.4 is a good example how experiments can suggest that an analysis can be sharpened.

This paper only scratches the surface of a related important methodological topic; namely how to perform experiments accurately and efficiently and how to evaluate the confidence in our findings statistically. In Sect. 4.2 we apply such a statistical test and find a very high level of confidence. In Sect. 4.4 we give an example how the number of repetitions can be coupled to the measured standard error. We also shortly discuss the choice of random number generator.

#### 4.1 Theory with Simplifications: Writing to Parallel Disks

Consider the following algorithm, EAGER, for writing  $D$  randomly allocated blocks of data to  $D$  parallel disks. EAGER is an important ingredient of a

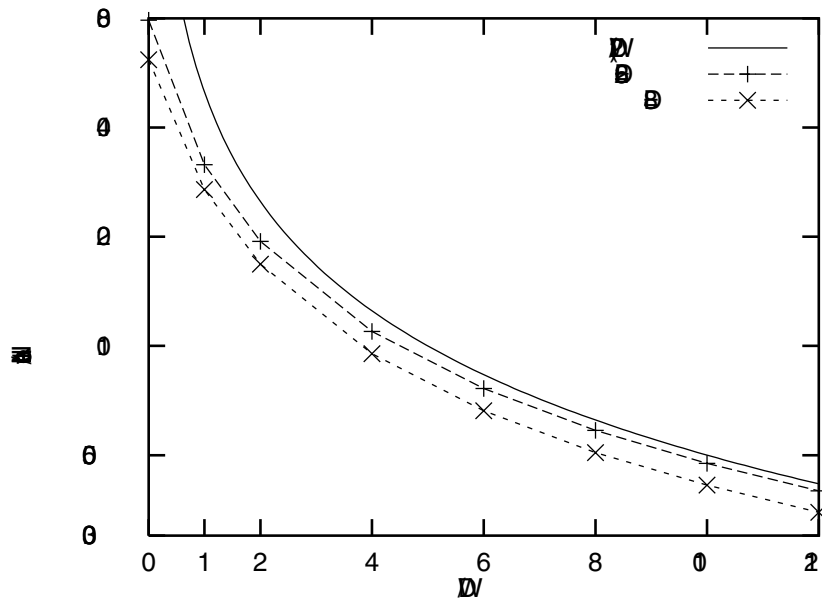


Fig. 1. Overhead (i.e., 1–efficiency) of EAGER.  $N = 10^6 \cdot D$  blocks were written.

general technique for scheduling parallel disks [18]. We maintain one queue  $Q_i$  for each disk. The queues share a buffer space of size  $W = \mathcal{O}(D)$ . We first put all the blocks into the queues and then write one block from each nonempty queue. If after that the sum of the queue lengths exceeds  $W$ , additional write steps are invested. We have no idea how to analyze this algorithm. Therefore, in [18] a different algorithm, THROTTLE, is proposed that only admits  $(1 - \epsilon)D$  blocks per time step to the buffers. Then it is quite easy to show using queuing theory that the expected sum of the queue lengths is  $D/(2\epsilon)$ . Further, it can be shown that the sum of the queue lengths is concentrated around its mean with high probability so that a slightly larger buffer suffices to make waiting steps rare.<sup>2</sup>

Still, in many practical situations EAGER is not only simpler but also somewhat more efficient. Was the theoretical analysis futile and misguided? One of the reasons why we think the theory is useful is that it suggests a nice explanation of the measurements shown in Fig. 1. It looks like  $1 - D/(2W)$  is a lower bound for the average efficiency of EAGER and a quite tight one for large  $D$ . This curve was not found by fitting a curve but by the observation that algorithm EAGER with  $\epsilon$  slightly larger than  $D/(2W)$  would yield a similar efficiency.

More generally speaking, the algorithms we are most interested in might be too difficult to understand analytically. Then it makes sense to analyze a related and possibly inferior algorithm and to use the scientific method to come to

<sup>2</sup> The current proof shows that  $W \in \mathcal{O}(D/\epsilon)$  suffices but we conjecture that this can be sharpened considerably using more detailed calculations.

theoretical insights about the original algorithm. In theoretical computer science, the latter step is sometimes omitted leading to friction between theory and practice.

#### 4.2 “Heuristic” Deduction: Random Polling

Let us consider the following simplified model for the startup phase of *random polling dynamic load balancing* [9,3,17] which is perhaps the best available algorithm for parallelizing tree shaped computations of unknown structure: There are  $n$  processing elements (PEs) numbered 0 through  $n - 1$ . At step  $t = 0$ , a random PE is busy while all other PEs are idle. In step  $t$ , a random shift  $k \in \{1, \dots, n - 1\}$  is determined and the idle PE with number  $i$  asks PE  $i + k \bmod n$  for work. Idle PEs which ask idle PEs remain idle; all others are busy now. How many steps  $T$  are needed until all PEs are busy? A trivial lower bound is  $T \geq \log n$  steps since the number of busy PEs can at most double in each step. An analysis for a more general model yields an  $\mathbf{E}[T] = \mathcal{O}(\log n)$  upper bound [17]. We will now argue that there is a much tighter upper bound of  $\mathbf{E}[T] \leq \log n + \log \ln n + 1$ .

Define the 0/1-random variable  $X_{ik}$  to be 1 iff PE  $i$  is busy at the beginning of step  $k$ . For fixed  $k$ , these variables are identically distributed and  $\mathbf{P}[X_{i0} = 1] = 1 - 1/n$ . Let  $U_k = \sum_{i < n} X_{ik}$ . We have

$$\mathbf{E}U_k = \mathbf{E} \sum_{i < n} X_{ik} = \sum_{i < n} \mathbf{P}[X_{ik} = 1] = n\mathbf{P}[X_{ik} = 1] .$$

Since the  $X_{ik}$  are not independent even for fixed  $k$ , we are stuck with this line of reasoning. However, if we simply assume independence, we get

$$\mathbf{P}[X_{i,k+1} = 0] = \mathbf{P}[X_{ik} = 0] \sum_{j \neq i} \frac{1}{n-1} \mathbf{P}[X_{jk} = 0] = \mathbf{P}[X_{ik} = 0]^2 ,$$

and, by induction,

$$\mathbf{P}[X_{ik} = 0] = (1 - 1/n)^{2^k} \leq e^{-2^k/n} .$$

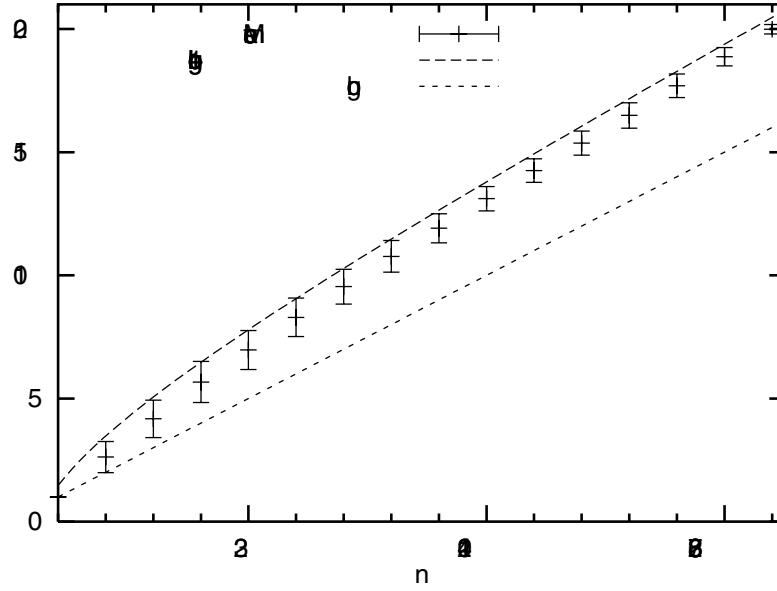
Therefore,  $\mathbf{E}[U_k] \geq n(1 - e^{-2^k/n})$  and for  $k = \log n + \log \ln n$ ,  $\mathbf{E}[U_k] \geq n - 1$ . One more step must get the last PE busy.

We have tested the hypothesis by simulating the process 1000 times for  $n = 2^j$  and  $j \in \{1, \dots, 16\}$ . Fig. 2 shows the results.

On the other hand, the measurements do exceed  $\log n + \log \ln n$ . We conjecture that our results can be verified using a calculation which does not need the independence assumption.

The probability that the measured values are only accidentally below the conjectured bound can be estimated using the Student- $t$  test. Following [13] we get a probability

$$1 - A \left( \sqrt{1000} \frac{\bar{T} - (\log n + \log \ln n + 1)}{\sigma} \middle| 999 \right)$$



**Fig. 2.** Number of random polling steps to get all PEs busy: Hypothesized upper bound, lower bound and measured averages with standard deviation.

where  $\bar{T}$  is the measured average,  $\sigma$  is the measured standard deviation and  $A(t|\nu)$  is the cumulative distribution function of the Student  $t$ -distribution with  $\nu$  degrees of freedom. Within the computational precision of Maple, this probability is zero.

### 4.3 Shellsort

Shellsort [20] is a classical sorting algorithm which is considered a good algorithm for almost sorted inputs in particular, if an in-place routine is desired or small to medium sized inputs are considered. Given an increasing integer sequence of offsets  $h_i$  with  $h_0 = 1$ , the following pseudo-code describes Shellsort.

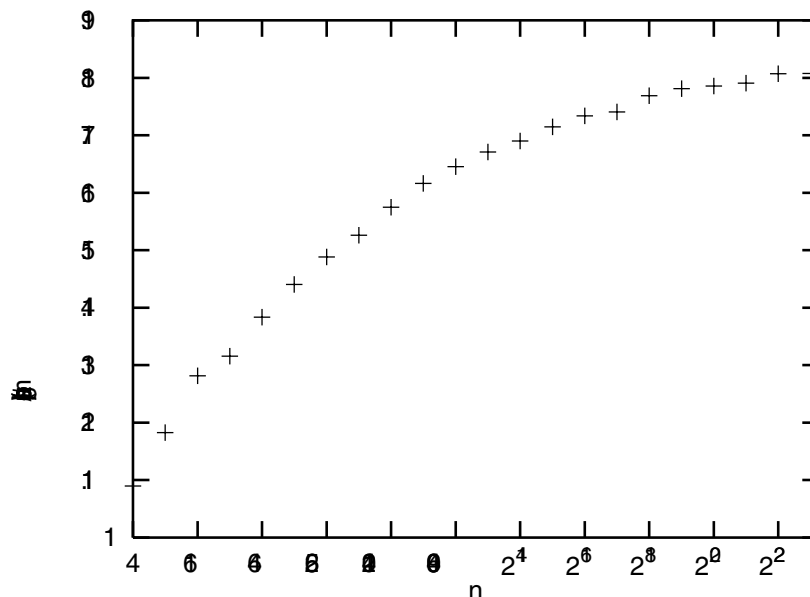
```

for each offset  $h_k$  in decreasing order do
  for  $j := h_k$  to  $n$  step  $h_k$  do
     $x := \text{data}[j]$ 
     $i := j - h_k$ 
    while  $i \geq 0 \wedge x < \text{data}[i]$  do
       $\text{data}[i + h_k] := \text{data}[i]$ 
       $i := i - h_k$ 
    od
     $\text{data}[i + h_k] := x$ 

```

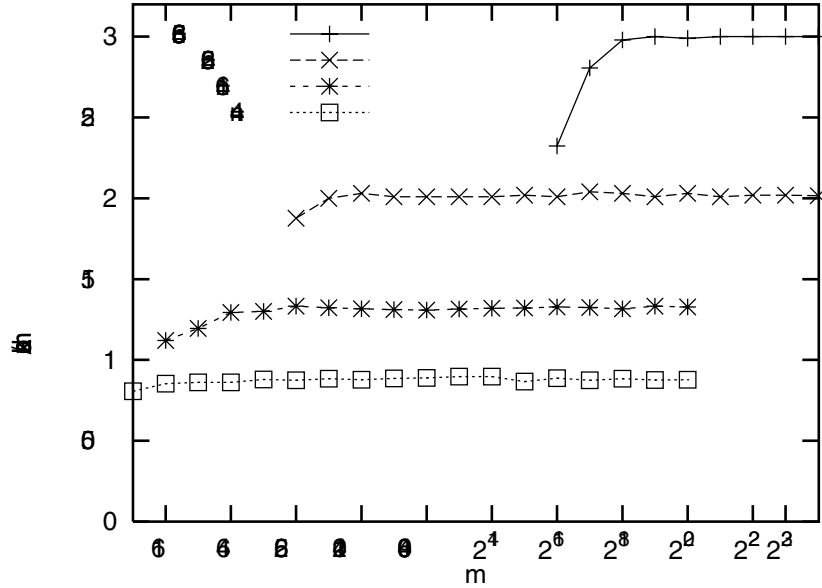
Interestingly, Shellsort still poses several open problems. For example, let  $T(n)$  denote the average number of key comparisons performed by Shellsort for  $n$

inputs. It is unknown whether there is an offset sequence which yields a sorting algorithm with  $T(n) = \mathcal{O}(n \log n)$  or even one with  $T(n) = o(n \log^2 n)$  [19,7]. It is known that any algorithm with  $T(n) = \mathcal{O}(n \log n)$  must use  $\Theta(\log n)$  offsets [7]. Previous experiments with many carefully constructed offset sequences led to the conjecture that no sequence yields  $T(n)$  close to  $\mathcal{O}(n \log n)$  [22].



**Fig. 3.** Competitive ratio of the average number of key comparisons of random offset Shellsort compared to the information theoretic lower bound  $\log(n!)$ . We used  $h_i := \lfloor h_{i-1} \cdot f_i + 1 \rfloor$  where  $f_i$  is a random factor from the interval  $[0, 4]$ . Averages are based on 1000 repetitions for  $n \leq 2^{13}$  and 100 repetitions for larger inputs.

Led by the successful use of randomness for sorting networks [10, Sect. 3.5.4] where no comparably good deterministic alternatives are known, we asked ourselves whether *random* offsets might work well for Shellsort. For our experiments we have used offsets which are the product of random numbers. The situation now is more difficult than in Sect. 4.2 where the theory gave us a very accurate hypothesis. Now we have little information about the dependence of the performance on  $n$ . Still, we should put the little things we do know into the measurements. First, by counting comparisons we can avoid the pitfalls of measuring execution time directly. Furthermore, we can divide these counts by the lower bound  $\log(n!) \approx n \log n - n / \ln(2)$  for comparison based sorting algorithms. The difficult part is to find an adequate model for the resulting quotient plotted in Fig. 3. According to the conjecture in [22] the quotient should follow a power law. In a semilogarithmic plot this should be an exponentially growing curve. So this conjecture is not a good model at least for realistic  $n$  (also remember



**Fig. 4.** Excess load for randomized balanced allocation as a function of  $n$  for different  $n$ . The experiments have been repeated at least sufficiently often to reduce the *standard error*  $\sigma/\sqrt{\text{repetitions}}$  [16] below one percent of the average excess load. In order to minimize artifacts of the random number generator, we have used a generator with good reputation and very long period ( $2^{19937} - 1$ ) [11]. In addition, we have repeated some experiments with the Unix generator `srand48` leading to almost identical results.

that Shellsort is usually *not* used for large inputs). A sorting time of  $\mathcal{O}(n \log^a n)$  for any constant  $a > 1$  would result in a curve converging to a straight line in Fig. 3. The curve gets flatter and flatter and its inclination might even converge to zero.

We might conjecture that  $T(n) = \mathcal{O}(n \log^{1+o(1)} n)$ . But we must be careful here. Because assertions like “ $T(n) = O(f(n))$ ” or “the inclination of  $g(n)$  converges to zero” are not experimentally falsifiable. One thing we could do however is to hypothesize that  $2^{T(n)/\log(n!)}$  is a concave function. This hypothesis is falsifiable and together with the measurements it implies<sup>3</sup>  $T(n) = \mathcal{O}(n \log^{1+\epsilon} n)$  for quite small values of  $\epsilon$  which we can further decrease by doing measurements for larger  $n$ .

#### 4.4 Sharpening a Theory: Randomized Balanced Allocation

Consider the following load balancing algorithm known as *random allocation*:  $m$  jobs are independently assigned to  $n$  processing elements (PEs) by choosing a

<sup>3</sup> We mean logical implication here, i.e., if the hypothesis is false nothing is said about the truth of the implied assertion.



target PE uniformly at random. Using Chernoff bounds, it can be seen that the maximum number of jobs assigned to any PE is

$$l_{\max} = m/n + \mathcal{O}\left(\sqrt{(m/n) \log n} + \log n\right)$$

with high probability (whp). For  $m = n$ ,

$$l_{\max} = \Theta(\log(n)/\log \log n)$$

whp can be proven.

Now consider the slightly more adaptive approach called *balanced random allocation*. Jobs are considered one after the other. Two random possible target PEs are chosen for each job and the job is allocated on the PE with lower load. Azar et al. [1] have shown that

$$l_{\max} = \mathcal{O}(m/n) + (1 + o(1)) \log \ln n$$

whp for  $m = n$ . Interestingly, this bound shows that balanced random allocation is exponentially better than plain random allocation. However, for large  $m$  their methods of analysis yield even weaker bounds than that for plain random allocation. Only very recently Berenbrink et al. [2] have shown (using quite nontrivial arguments) that

$$l_{\max} = m/n + (1 + o(1)) \log \ln n .$$

Fig. 4 shows that a simple experiment at least predicts that  $l_{\max} - m/n$  cannot depend much on  $m$ . Other researchers (e.g. [8]) made some experiments but without trying to induce hypotheses on the asymptotic behavior of balanced allocation.

Our experiments were done before the theoretical solution. Otherwise, we could have picked one of the other open problems in the area of balls into bins games. For example, Vöcking [21] recently proved that an asymmetric placement rule for breaking ties can significantly reduce  $l_{\max}$  for  $m = n$  but nobody seems to know how to generalize this result for general  $m$ .

## 5 Discussion

Assume that using the scientific method we have found an experimentally well supported hypothesis about the running time of an important, difficult to analyze algorithm. How should this result be interpreted? It may be viewed as a conjecture for guiding further theoretical research for a mathematical proof. If this proof is not found, a well tested hypothesis may also serve as a surrogate. For example, in algorithmics the hypotheses “a good implementation of the simplex method runs in polynomial time” or “NP-complete problems are hard to solve in the worst case” play an important role. The success of the scientific method in the natural sciences — even where deductive results would be possible in principle — is a further hint that such hypotheses may play an increasingly

important role in algorithmics. For example, Cohen-Tannoudji et al. [4] (after 1095 pages of deductive results) state that “in all fields of physics, there are very few problems which can be treated completely analytically.” Even a simple two-body system like the hydrogen atom cannot be handled analytically without making simplifying assumptions (like handling the proton classically). For the same reason, experiments are of utmost importance in chemistry although there is little doubt that well known laws like the Schrödinger equation in principle could explain most of chemistry.

## References

1. Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *26th ACM Symposium on the Theory of Computing*, pages 593–602, 1994.
2. P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. In *32th Annual ACM Symposium on Theory of Computing*, 2000.
3. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science*, pages 356–368, Santa Fe, 1994.
4. C. Cohen-Tannoudji, B. Diu, and F. Laloë. *Quantum Mechanics*, volume 2. John Wiley & Sons, Inc., 1977.
5. A. Goldberg and B. Moret. Combinatorial algorithms test sets (cats). In *10th ACM-SIAM Symposium on Discrete Algorithms*, 1999.
6. J. Hooker. Needed : An empirical science of algorithms. *Operations Res.*, 42(2):201–212, 1994.
7. T. Jiang, M. Li, and P. Vitányi. Average-case complexity of shellsort. In *ICALP*, number 1644 in LNCS, pages 453–462, 1999.
8. J. Korst. Random duplicate assignment: An alternative to striping in video servers. In *ACM Multimedia*, pages 219–226, Seattle, 1997.
9. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.
10. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.
11. M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998. <http://www.math.keio.ac.jp/~matumoto/emt.html>.
12. C. C. McGeoch, D. Precup, and P. R. Cohen. How to find big-oh in your data set (and how not to). In *Advances in Intelligent Data Analysis*, number 1280 in LNCS, pages 41–52, 1997.
13. P. H. Müller. *Lexikon der Stochastik*. Akademie Verlag, 5th edition, 1991.
14. R. Niedermeier, K. Reinhard, and P. Sanders. Towards optimal locality in mesh-indexings. In B. S. Chlebus and L. Czaaja, editors, *Fundamentals of Computation Theory*, number 1279 in LNCS, pages 364–375, Krakow, 1997.
15. K. R. Popper. *Logik der Forschung*. Springer, 1934. English Translation: *The Logic of Scientific Discovery*, Hutchinson, 1959.
16. W. H. Press, S. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2. edition, 1992.
17. P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.

18. P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *11th ACM-SIAM Symposium on Discrete Algorithms*, pages 849–858, 2000.
19. R. Sedgewick. Analysis of shellsort and related algorithms. *LNCS*, 1136:1–11, 1996.
20. D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–33, July 1958.
21. B. Vöcking. How asymmetry helps load balancing. In *40th FOCS*, pages 131–140, 1999.
22. M. A. Weiss. Empirical study of the expected running time of shellsort. *The Computer Journal*, 34(1):88–91, 1991.

# Visualizing Algorithms Over the Web with the Publication-Driven Approach<sup>\*</sup>

Camil Demetrescu<sup>1</sup>, Irene Finocchi<sup>2</sup>, and Giuseppe Liotta<sup>3</sup>

<sup>1</sup> Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”,  
Via Salaria 113, 00198 Roma, Italy  
Tel. +39-06-4991-8442, [demetres@dis.uniroma1.it](mailto:demetres@dis.uniroma1.it)

<sup>2</sup> Dipartimento di Scienze dell’Informazione, Università di Roma “La Sapienza”,  
Via Salaria 113, 00198 Roma, Italy  
Tel. +39-06-4991-8430, [finocchi@dsi.uniroma1.it](mailto:finocchi@dsi.uniroma1.it)

<sup>3</sup> Dipartimento di Ingegneria Elettronica e dell’Informazione, Università di Perugia,  
Via G. Duranti 93, 06125 Perugia, Italy  
Tel. +39-075-5852685, [liotta@diei.unipg.it](mailto:liotta@diei.unipg.it)

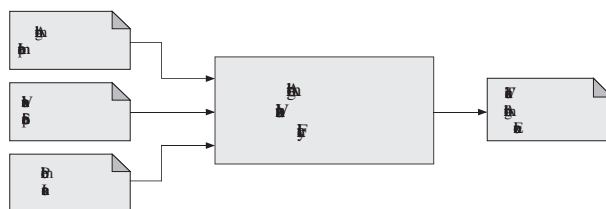
**Abstract.** We propose a new approach to algorithm visualization over the Web, called *publication-driven approach*. According to the publication-driven approach, algorithms run on a developer’s remote server and their data structures are published on blackboards held by the clients. Animations are specified by attaching visualization handlers to the data structures published on the client’s blackboard: modifications to these structures, due to the remote algorithm execution, trigger the running of the corresponding handlers on the client’s side. The publication-driven approach has been used for designing an algorithm visualization facility over the Web, called *WAVE*. A first prototype of *WAVE* is available at the URL <http://www.dis.uniroma1.it/~wave>.

## 1 Introduction

Algorithm visualization concerns the high-level graphical display of both the control flow and the data flow of running programs [16]. The visualization of an algorithm, and in particular the pictorial representation of the evolution of its data structures, has several applications both in the algorithm engineering and in the educational field [2,9]. Indeed, a complex algorithm can be described not only by presenting a commented piece of code, but also by associating geometric shapes with the algorithm’s data and by showing how these shapes move and change as the algorithm implementation is executed. The latter capability is not only an effective tool for algorithms researchers who want to share and disseminate their ideas, but also provides a valuable help for discovering degeneracies, i.e., special cases for which the algorithm may not produce a correct

---

<sup>\*</sup> Work partially supported by the project “Algorithms for Large Data Sets: Science and Engineering” of the Italian Ministry of University and Scientific and Technological Research (MURST 40%) and by the project “Geometria Computazionale Robusta con Applicazioni alla Grafica e al CAD” of the Italian Research Council.



**Fig. 1.** Algorithm visualization facility seen as a “black box”.

output. Moreover, teachers may benefit from the support of algorithm visualization to describe the behaviour of an algorithm with a more attractive and communicative medium than the traditional blackboard.

A lot of effort has been put in recent years to designing and developing *algorithm visualization facilities* (see, e.g., the systems reviewed in [11,16] and [2,4,5,6,7,8,10,17]). An algorithm visualization facility, seen as a “black box”, is shown in Figure 1: it receives as input the implementation of an algorithm, an instance of the input of the algorithm, and a specification of the visualization of the algorithm, that is a mapping from the set of algorithms data to a set of geometric shapes that will be displayed; the output is a *visual trace* of the algorithm execution, i.e., a sequence of moving images that describe the evolution of the data at-run-time.

There is a general consensus that algorithm visualization facilities can strongly benefit from the potentialities offered by the World Wide Web. Indeed, the use of the Web for easy communication, education, and distance learning can be naturally considered a valid support for improving the cooperation between students and instructors, and between algorithm engineers. Valuable features of an effective Web-based algorithm visualization facility are:

**Task Decoupling:** The algorithm visualization facility should help with separating the roles of *Algorithm Designer*, *Algorithm Developer*, *Algorithm Visualizer*, and *End-user* [16], and with making their interaction easier and faster.

**Language Independence:** Algorithm developers usually want to implement algorithms in their favourite programming language. Therefore, an effective algorithm visualization facility should be able to accept programs written in different languages and to allow the algorithm visualizer to design a visualization independently of the algorithm implementation language.

**Customizability:** Algorithm visualizers should be given the possibility of customizing the visualization and to design different graphical interpretations of the same data structures without having neither a detailed knowledge of the source code nor ability with computer graphics concepts and libraries.

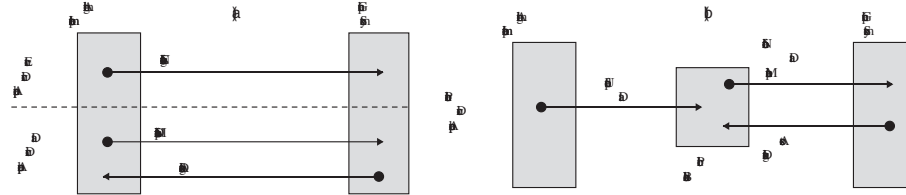
**Code Protection:** Since developers may want to keep privacy and protection on their code, visualizations should be designed without downloading it.

**User-Friendliness:** It should be easy to learn how to use the algorithm visualization facility for all the different types of users. The interaction with the visualization facility should be supported by a friendly environment.

The simplest approach to algorithm animation over the WEB consists of exploiting directly the graphic capabilities offered by the Java language without making use of any algorithm animation facility. In particular, it is very common to find out Java applets over the WEB consisting of the implementation of a specific algorithm annotated with hooks to Java visualization methods. Several examples can be found at the URL <http://www.cs.hope.edu/~algaanim/ccaa/site.html>. Such animations usually portray the execution of the algorithm on predefined test sets or on problem instances specified by the end user that interacts with the applet. Unfortunately, although some of these animations are of very good quality due to the high level of customizability offered by Java, this approach has several disadvantages with respect to the above features: it is very time consuming, requires algorithms to be written in Java, guarantees no protection of the code, and assumes the algorithm developer to be the same person as the algorithm visualizer. Even those systems explicitly designed to support visualization over the Web are either targeted for particular classes of algorithms, or do not completely satisfy the aforementioned requirements. In the following we give a few examples.

JEliot [10] aims at making automatic the task of producing visualizations of Java programs. It parses the Java code and allows the user to choose the cast of variables to visualize on the stage according to built-in graphical interpretations. Changes to the variables are automatically detected as the program runs and the visualization is coherently updated. JEliot relieves the user from writing any visualization code and so it is very easy to use, but does not support a high level of customizability. JDSL [2] is a library of data structures in Java which features a visualizer for animating operations on abstract data types such as AVL Trees, Heaps, and Red-Black Trees. It is well suited for educational purposes as students are allowed to write and test their own classes provided they implement specific JDSL Java interfaces. In particular, constructors and selectors of the data types must be implemented as methods having predefined signatures. This allows JDSL visualizer to invoke them as the student interacts with the system for testing her implementation. The visualization of supported data types is embedded into the library. VEGA [12] is a C++ client/server visualization environment especially targeted for geometric algorithms. The algorithm is executed on the server; the client runs on any Java Virtual Machine and a small bandwidth communication interface guarantees good performance even in slow networks. The end-user can interactively draw, load, save, and modify graphical scenes, can visualize algorithms on-line or show saved runs off-line, and can customized the visualization by specifying a suitable set of view attributes.

In this paper we propose a new approach to algorithm visualization on the Web, called *publication-driven approach*. According to the publication-driven approach, algorithms run on a developer's remote server, while their visualization can be realized and viewed through any Java-enabled Web-browser. The visualization is specified with respect to an intermediate set of *public* data structures that are a coherent copy of a subset of the program's data structures. In the following we call this intermediate set of data structures *public blackboard*. With the publication-driven approach, an algorithm visualization is realized by performing two steps: (1) a set of program's variables, containing all the information



**Fig. 2.** (a) Event-driven and data-driven approaches to visualization; (b) publication-driven approach.

to be conveyed into the visualization, is chosen for publication on the public blackboard; (2) visualization handlers are attached to the variables of the public blackboard and are executed whenever the variables themselves change. These handlers may be chosen from a library of predefined ones, yet experienced users are allowed to write new handlers in a suitable visualization language. Modifications to the variables in the public blackboard, due to the remote algorithm's execution, trigger the running of the corresponding handlers on the client's side. We remark that a concept similar to that of public blackboard has been used in the field of computational steering where, in the traditional cycle of a simulation (i.e., preparing input, executing a simulation, and visualizing the results as a post-processing step), researchers change parameters of their simulations on the fly and immediately receive feedback on the effect [13].

The publication-driven approach has been used to design the architecture of an algorithm visualization facility called *WAVE* (Web Algorithm Visualization Environment). A first prototype of the system is currently being implemented and experimented. The interested reader can find further information at the URL <http://www.dis.uniroma1.it/~wave>.

## 2 The Publication-Driven Approach

In this section we describe the main ideas that underlie the publication-driven approach and compare it with other existing approaches. As shown in Figure 2(a), the approaches to algorithm visualization described in the literature can be roughly classified into two main groups, i.e., *event-driven* and *data-driven* approaches, according to the way the visualization events are triggered and the information to be visualized are gathered. Furthermore, systems that allow users to customize the visualization by means of a suitable language can be classified according to the paradigm of the visualization language, i.e., imperative, declarative, constraint-based, or object-oriented.

The *event-driven approach* was pioneered by BALSA [5] and has been subsequently followed by many other systems: visualizations are realized by identifying *interesting events* in the source code that correspond to relevant steps of the underlying algorithm, and by annotating the code with calls to visualization routines of the graphic system (these routines are usually written in imperative or object-oriented languages). The event-driven approach allows the algorithm

visualizer to easily customize the visualization according to her needs and appears simple to understand; however, even the visualization of moderately easy and short programs may require several lines of additional code.

In the *data-driven approach* visualization events are triggered by modifications to the content of the data structures performed by the program during its execution. The main motivation behind this approach is that observing how the data change during the execution of an algorithm is usually enough to understand the algorithm behavior. The data-driven approach relieves the algorithm visualizer from the task of identifying interesting events and makes it possible to concentrate only on the data structures of the algorithm, thus achieving high levels of abstraction and automation. In spite of its flexibility, only few algorithm visualization facilities have explored its potentialities. Existing systems are either automatic [10] or based on a declarative [15] or logic-based [7] specification of the visualization.

WAVE is a data-driven imperative algorithm visualization facility. We call the approach that underlies WAVE *publication-driven approach*. As shown in Figure 2(b), the publication-driven approach introduces an software layer, called *public blackboard*, between the algorithm implementation and the graphic system. The public blackboard maintains a copy of those data structures that are the subject of the visualization. Each time a data is modified by the program, the copy of this data in the public blackboard is coherently updated. Also, the public blackboard notifies the update to the graphic system, which accesses the modified data in the public blackboard itself and updates the visualization accordingly.

A data structure that has a copy on the public blackboard is said to be *published* on the blackboard. Data structures are published by means of an *algorithm publication mechanism*, which consists of annotating the source code with meaningful events (i.e., variable declarations, changes, and going out of scope) and of mirroring such events on the public data. The annotation of the source code is the task of the *algorithm publisher*, that thus decouples the role of the algorithm visualizer from that of the algorithm developer. The *algorithm visualization mechanism* consists of attaching visualization handlers to the public data structures. The *algorithm visualizer* accomplishes the task of designing these visualization handlers and of associating them with the public data structures. In the sequel we provide some more details on these mechanisms and give simple examples. A more elaborate example is discussed in Section 4.

**The Algorithm Publication Mechanism** The publication mechanism exploits a suitable publication language whose main goal is to encode *execution traces* instead of the program itself: execution traces are plain descriptions of the published data structures and of their manipulations when the program is executed on a given input. Such traces are the only information about the program that are delivered over the Web, which guarantees a high level of code protection. In a trace, *declarations* of strongly typed public data structures alternate with *assignment* statements and with *un-declarations*. Declarations are introduced by the keyword “**declare**” followed by the name of the variable, by its type, and by its address in main memory. This last information makes it possible to handle dynamic variables and pointers arithmetic. Assignment statements use the operator “**:=**” and follow a Pascal-like syntax. Un-declarations consist of the name





	
<pre> int i; struct {int x; float y;} v[100];  void main() {     for (i=0; i&lt;100; i++)         v[i].x=v[i].y=i*2; } </pre>	<pre> int i; struct {int x; float y;} v[100];  void main() {     printf("declare k:integer;");     printf("declare a:array[100] of integer;");     for (i=0; i&lt;100; i++) {         printf("i:=%d;", i);         v[i].x=v[i].y=i*2;         printf("a[%d]:=%d;", i, v[i].x);     }     printf("undeclare a;");     printf("undeclare k;"); } </pre>

Fig. 3. Original and annotated fragment of C code.

of the variable prefixed by the keyword “undeclare” and allow the simulation of the scope rules of the published data structures. Execution traces are generated by capturing the standard output stream of the program, that is sent over the Web for visualization. The link between a program variable and its copy on the public blackboard is created by interpreting these printing statements.

As an example, let us consider a simple C program that fills in an array of pairs of numbers and the same program annotated for publication as shown in Figure 3. In the annotated code, we declare two public variables, named **k** and **a**, associated with the control variable **i** and with the array **v** of the original code, respectively. Observe that only the integer field of array **v** is published, showing that the publication step can be useful for omitting details irrelevant for the comprehension of the algorithm. The values of **k** and **a** are updated each time the values of **i** and **v** change; **k** and **a** are finally un-declared when **i** and **v** go out of scope.

**The Algorithm Visualization Mechanism** As for the publication mechanism also the visualization mechanism is based on a suitable language, whose syntax is very similar to that of the publication language. The algorithm visualizer designs a visualization of a program by performing the following two steps: (1) A set of *visualization handlers* is written in the visualization language. A visualization handler is a routine implemented in the visualization language that can access the variables on the public blackboard and invoke 2-D graphical primitives that draw basic geometric objects, such as disks, lines, points, and so on. (2) The visualization handlers are associated with the events of declaration, assignment, and un-declaration of public variables. This is done by defining a set of *associations variables-handlers*. When a remote manipulation of a variable of the public blackboard takes place, the corresponding visualization handler is automatically executed by the system.

As an example, consider the public variable **k** of Figure 3 and suppose that the algorithm visualizer designs a visualization where there is a disk whose radius is equal to the value of **k**. Figure 4 shows the three visualization handlers (i.e., **CreateCircle**, **ResizeCircle**, and **DeleteCircle**) and the three associations variables-handlers that are needed for this task.

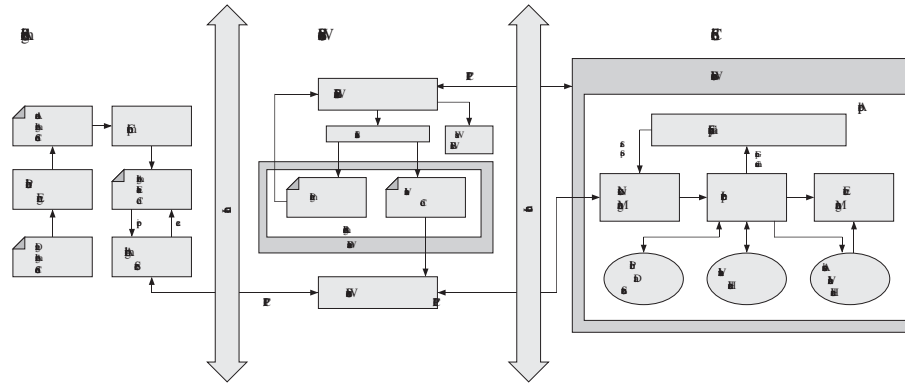
■	
<pre> function ResizeCircle() begin   call "size Circle 1 "+a[0]+" 1"   call "update"; end;  function CreateCircle() begin   call "create Circle 1 1";   call "locate Circle 1 100 100 1";   call ResizeCircle(); end; </pre>	<pre> function DeleteCircle() begin   call "delete Circle 1 1"; end;  declare a:    call CreateCircle(); change a:    call ResizeCircle(); undeclare a: call DeleteCircle(); </pre>

**Fig. 4.** Example of handlers and associations variables/handlers for visualizing the program of Figure 3.

### 3 The Architecture of WAVE

The architecture of WAVE is shown in Figure 5. The interaction between clients and the system takes place through any Java-enabled Web browser and is based on the so called Mocha model [1,3]: the user interacts with a hypertextual interface which is transferred over the Web to the user's machine together with the code for the interface itself. In this model the visualization is displayed by an applet that runs locally, while the algorithm is executed on a remote server, thus guaranteeing good code protection for the source code and reducing the load on the network. As Figure 5 shows, WAVE works on three sides:

- **Algorithm Developer Side.** Algorithm implementation codes are annotated through a *Publication Engine* (further described in Section 4) and are compiled on the developer's local machine. An *Algorithm Server*, started by the local system administrator, runs the resulting executable codes on client's demand. The standard output of executed programs consists of scripts in the publication language that declare, manipulate, and un-declare variables of the public blackboard, and is sent by the algorithm server to the remote client.
- **WAVE Server Side.** It contains a *Web site*, that represents the Web front-end of WAVE Server Side, and a *Database*, consisting of a collection of *Algorithm Records*. Each Algorithm Record is associated with an algorithm whose visualization has been designed and made available. The Algorithm Record stores a description of the algorithm together with the set of visualization handlers and the set of associations variables-handlers that are needed for its animation. Finally, the *WAVE Server* acts as a bridge between the Algorithm Server and the Client Side and provides both the visualization handlers and the associations variables-handlers for a specific algorithm chosen from WAVE Database.
- **Client Side:** Users interact with WAVE by running a general-purpose applet, called *WAVE Applet* embedded into ad-hoc Web pages dynamically generated by WAVE Web Server. The main modules of *WAVE applet* are: a *Graphic System*, which allows end-users to stop, restart, pause, step, and continue a visualization and provides 2D graphic capabilities; a *Network*



**Fig. 5.** The architecture of WAVE.

*Manager*, which allows WAVE Applet to make connections over the Internet; an *Interpreter*, which parses and executes the statements in the publication and visualization languages, accesses information in the public blackboard, and issues graphic commands to the GUI; and finally an *Event Manager*, which checks if execution traces coming from the Algorithm Server imply the execution of visualization handlers according to the associations variables-handlers.

The interaction between WAVE and the users changes depending on the different types of users. When an algorithm developer wants to make a new algorithm implementation available for visualization, she downloads from WAVE's web site the Publication Engine, which parses the code and asks the developer for those data that have to be published. Also, the Publication Engine automatically generates an annotated version of the source code, that is then compiled by the developer and made available on the Algorithm Server, which runs on the local Server of the developer. The developer is also requested to provide information about the new algorithm, such as the name of the algorithm, the Internet address/port number of the Algorithm Server, and the algorithm executable file pathname. All these information are stored in an algorithm record of WAVE Database associated with the new algorithm. The design of a visualization for the new algorithm is now the task of the algorithm visualizer, who adds visualization handlers and associations variables-handlers in the Algorithm Record. The visualization handlers can be either chosen among those of a Visualization Library, or can be new handlers designed ad-hoc for the specific animation. There can be more than one animation for a given algorithm. Finally, end-users can access the list of algorithms that are available for visualization in the WAVE Database and select one of them. End-users are given the possibility of choosing among the different possible visualizations that have been designed for an algorithm and also can customize the chosen animation by changing some parameters of the visualization handlers. For example, they can change color, thickness, or shape for a geometric object specified in a visualization handler. The next section illustrates an example of interaction between Wave and its users.

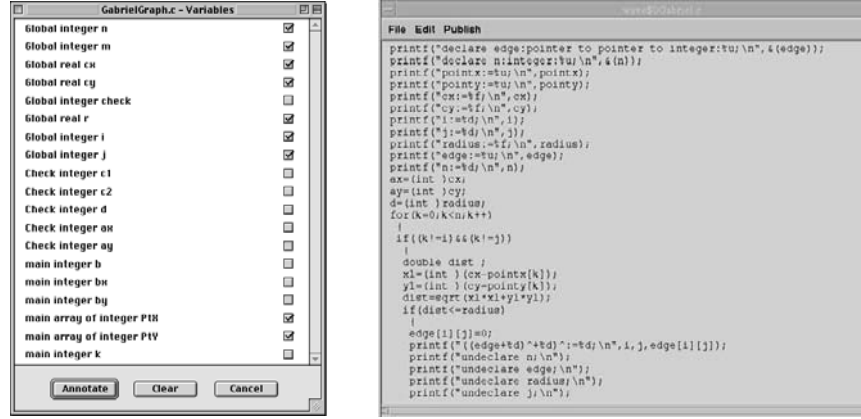


Fig. 6. (a) The Publication Dialog; (b) an excerpt of annotated code.

## 4 Interacting with WAVE

Based on the architecture described in Section 3 we have developed a prototype of WAVE, currently available at the URL <http://www.dis.uniroma1.it/~wave>. The example of interaction with the prototype of WAVE that we are going to show concerns the visualization of an algorithm for computing the *Gabriel graph* of a set of points in the plane [14]. Given a set  $S$  of distinct points in the plane, the Gabriel graph of  $S$  is a geometric graph whose vertices are the elements of  $S$  and such that there is an arc between two vertices  $P_1$  and  $P_2$  if and only if the disk whose diameter is the segment  $P_1P_2$  does not contain any other element of  $S$ . In our example, we refer to a brute-force algorithm for computing the Gabriel graph of  $S$  that checks all pairs of vertices to define the edges. A possible visualization consists of showing for each edge found by the algorithm the corresponding disk. The algorithm has been implemented in C, which is the language supported by the Publication Engine. When the developer submits the code to the Publication Engine, the latter returns the Publication Dialog shown in Figure 6(a) with the list of variables used in the program. The developer can now choose which variables she wants to be published and described in the program's execution trace. In our example, we assume that the developer selects, among others, the following variables: the number of points  $n$ ; the number of arcs  $m$  (the value of  $m$  is originally equal to 0, but is progressively incremented as the algorithm runs); two arrays of size  $n$ ,  $PtX$  and  $PtY$ , containing the  $x$ -coordinates and the  $y$ -coordinates of the points, respectively; the indices  $i$  and  $j$  that refer to the pairs of points that are progressively considered by the algorithm (the value of these indices range from 0 to  $n - 1$ ); the coordinates  $cx$  and  $cy$  and the radius  $r$  of the circle considered by the algorithm at each step. After the selection of public variables, the Publication Engine parses again the code and automatically annotates it with publication statements. Figure 6(b) shows an excerpt of annotated code, that can be finally compiled.

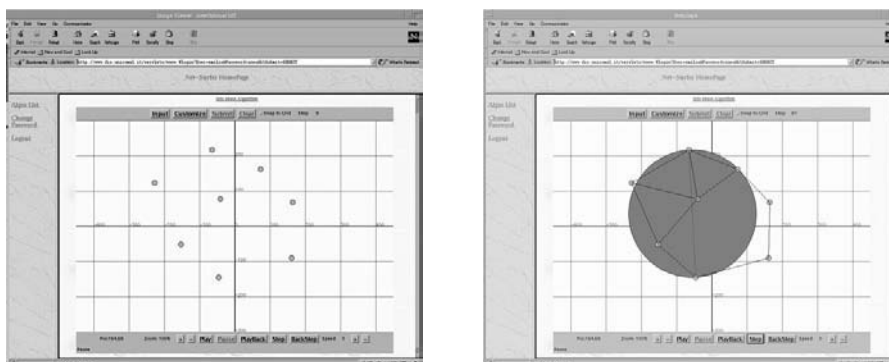
M	
<pre> function CreateCircle(c:integer) begin   call "create Circle " +c+ " 1";   call "locate Circle " +c+ " +cx+ " +cy+ " 1";   call "size Circle " +c+ " +r+ " 1";   call "color Circle " +c+ " 1 0 0 1";   call "layer Circle " +c+ " 0 1";   call "update"; end;  function EraseCircle(c:integer) begin   call "delete Circle " +c+ " 1"; end;  function CreatePoints(nodes:integer) begin   if(nodes&gt;1) then call CreatePoints(nodes-1);   call "create Point_ " +nodes+ " 1"; end; </pre>	<pre> function LocatePoint(node:integer) begin   call "locate Point_ " +node+ " " +     PtX[node]+ " " +     PtY[node]+ " 1";   call "layer Point_ " +node+ " 1 1";   call "update"; end;  function AddArc(arc:integer; start:integer; end:integer) begin   call "create Line " +arc+ " 1";   call "locate Line " +arc+ " +PtX[start]+ " +     PtY[start]+ " 1";   call "size Line " +arc+ " +PtX[end]+ " +     PtY[end]+ " 1";   call "layer Line " +arc+ " 2 1";   if(arc&gt;0) then call EraseCircle(arc-1);   call CreateCircle(arc);   call "update"; end; </pre>
M	
<pre> declare PTX; change PTY; undeclare PTY; change m; </pre>	<pre> call CreatePoints(n); call LocatePoint(i); call EraseCircle(s); call AddArc(s,i,j); </pre>

Fig. 7. Visualization code related to the public variables of Figure 6.

The task of the algorithm visualizer is now to design the visualization of the program by defining visualization handlers and associations variables-handlers for the variables that have been selected in the Publication Dialog. An example of Visualization handlers and Associations variables-handlers is shown in Figure 7. The visualization handlers in Figure 7 are as follows:

- **CreatePoints** creates a set of points with cardinality equal to the value of the input parameter *nodes*. These points are numbered from 0 to *nodes* – 1.
- **LocatePoint** locates in  $(PtX[node], PtY[node])$  the point identified by the number *node*. The point is also assigned with a layer that controls its visibility with respect to other graphical objects covering the same position.
- **AddArc** creates a new line. The line is identified by number *arc* and goes from point *start* (located in  $(PtX[start], PtY[start])$ ) to point *end* (located in  $(PtX[end], PtY[end])$ ). This function also erases the circle drawn at the previous step of the program's execution (if any), and draws the new circle.
- **DrawCircle** creates a new circle centered in  $(cx, cy)$  and having radius equal to *r*. The circle is colored red.
- **EraseCircle** erases the circle identified by the number *c*.

The visualization is now ready to be run by an end-user. Figure 8 describes the interaction between the end-user and WAVE. Figure 8(a) shows a WAVE Applet with a canvas that allows the User to graphically define a set of points. The applet provides basic features for editing the input, such as snapping the points to the grid, moving them on the canvas, zooming in and out, and saving the input on a file. Figure 8(b) shows the running visualization: at any pair of points considered by the program there corresponds the visualization of a disk on the canvas. If the disk does not contain any other points, then it is colored green and the segment connecting the points is chosen to be an edge of the Gabriel graph; otherwise, the disk is colored red and the segment connecting the points is not an edge of the Gabriel graph. When the first run is terminated, the end-user can play back and forth the visualization and can review it proceeding in a step-by-step mode. Also, the end-user is given the capability of customizing the



**Fig. 8.** (a) A WAVE Applet for editing the input of a program that computes a Gabriel Graph. (b) Running the visualization of the Gabriel graph program.

visualization by clicking on the “customize” button and editing the visualization handlers; a possible customization for this example consists of changing the color of the disks.

## 5 Conclusions and Future Work

In this paper we have presented a new approach to algorithm visualization over the Web, called publication driven approach. We have also described the architecture and the prototype of WAVE, an algorithm visualization facility based on the publication-driven approach. The current prototype of WAVE has supported our research showing the feasibility of the publication-driven approach.

There are still several features that we would like to implement in the system and some design issues that we would like to explore. In particular, we would like to extend the publication engine, which currently supports C, with the possibility of supporting C++ and Java in a near future. Within the object-oriented paradigm, it could be also interesting to explore other ways of implementing the publication mechanism, e.g., encapsulating publication statements into classes. We finally planned to study the visualization of special families of algorithms, such as graph drawing algorithms.

## Acknowledgements

We would like to thank E. Di Giacomo who has been cooperating with us in the development of the current prototype of WAVE.

## References

1. J.E. Baker, I. Cruz, G. Liotta, and R. Tamassia. Visualizing Geometric Algorithms over the Web. *Computational Geometry: Theory and Applications*, 12, 1999.

2. R.S. Baker, M. Boilen, M.T. Goodrich, R. Tamassia, and B. Stibel. Testers and Visualizers for Teaching Data Structures. *ACM SIGCSE Bulletin*, 31, 1999.
3. G. Barequet, S. Bridgeman, C. Duncan, M.T. Goodrich, and R. Tamassia. Geom-Net: Geometric Computing over the Internet. *IEEE Internet Computing*, 1999.
4. M.H. Brown and M. Najork. Collaborative Active Textbooks: a Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 12th IEEE Symposium on Visual Languages (VL'96)*, pages 266–275, 1996.
5. M.H. Brown and R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3):177–186, 1984.
6. G. Cattaneo, U. Ferraro, G.F. Italiano, and V. Scarano. Cooperative Algorithm and Data Types Animation over the Net. In *Proc. XV IFIP World Computer Congress, Invited Lecture*, pages 63–80, 1998.
7. P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2), 2000. Leonardo is available at the URL <http://www.dis.uniroma1.it/~demetres/Leonardo/>.
8. C. Demetrescu and I. Finocchi. Smooth Animation of Algorithms in a Declarative Framework. In *Proceedings of the 15th IEEE Symposium on Visual Languages (VL'99)*, pages 280–287, 1999.
9. M.T. Goodrich and R. Tamassia. Teaching the Analysis of Algorithms with Visual Proofs. In *ACM SIGCSE '98*, 1998.
10. J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of User Algorithms on the Web. In *Proceedings of the 13th IEEE Symposium on Visual Languages (VL'97)*, pages 360–367, 1997.
11. A. Hausner and D. Dobkin. Making Geometry Visible: an Introduction to the Animation of Geometric Algorithms. In *Handbook for Computational Geometry*, edited by Sack and urrutia.
12. Ch.A. Hipke and S. Schuierer. VEGA: A User Centered Approach to the Distributed Visualization of Geometric Algorithms. In *Proceedings of the 7-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99)*, pages 110–117, 1999.
13. J. Mulder, J. van Wijk, and R. van Liere. A Survey for Computational Steering Environments. *Future Generation Computer Systems*, 15:2, 1999.
14. F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer Verlag, 1985.
15. G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y. Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.
16. J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1997.
17. A. Tal and D. Dobkin. Visualization of Geometric Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):194–204, 1995.

# Interchanging Two Segments of an Array in a Hierarchical Memory System

Jesper Bojesen<sup>1</sup> and Jyrki Katajainen<sup>2</sup>

<sup>1</sup> UNI-C, Danish Computing Centre for Research and Education,  
Technical University of Denmark, Building 304,  
DK-2800 Lyngby, Denmark

<sup>2</sup> Department of Computing, University of Copenhagen,  
Universitetsparken 1, DK-2100 Copenhagen East, Denmark

**Abstract.** The problem of interchanging two segments of an array is considered. Using the known methods as a starting-point, two new adaptations are developed that achieve higher memory locality. It is confirmed, both analytically and experimentally, that on a computer with a hierarchical memory the adaptations are superior to the earlier methods.

## 1 Problem Statement

Assume that we are given an array  $\mathbf{A}$  of size  $N$ . Let  $\alpha = x_0x_1 \dots x_{m-1}$  and  $\beta = x_mx_{m+1} \dots x_{m+n-1}$  be two adjacent **segments** of  $\mathbf{A}$  such that  $N = m+n$ . In the **segment-interchanging** problem the task is to change the contents of  $\mathbf{A}$  from  $\alpha\beta = x_0x_1 \dots x_{m+n-1}$  to  $\beta\alpha = x_mx_{m+1} \dots x_{m+n-1}x_0x_1 \dots x_{m-1}$ . Alternatively, the problem can be formulated as follows: given an array  $\mathbf{A}$  of size  $N$ , rotate all the elements of  $\mathbf{A}$  (simultaneously)  $n$  positions to the right (mod  $N$ ) [or equivalently  $m$  positions to the left (mod  $N$ )]. As an additional requirement, the interchange should be carried out as space-efficiently as possible. For the sake of simplicity, we assume that the array elements are integers, each of which can be stored in a single computer word.

In the literature the problem has been discussed at least under the names: block interchange [5,11,12], block transposition [10], section swap [6,7], vector exchange [3], vector rotation [2], segment exchange [4], and array rotation [13]. In our choice **segment interchange**, the selection of the verb **interchange** was quite arbitrary, but we use the term **segment** to avoid the collision with a separate concept **block** that means an entity in which data is transferred from one memory level to another. In our analysis the term **array** means any sequence of elements stored in consecutive locations in memory, even though most methods can handle more general sequences. Moreover, we shall use the term **segment swapping** to denote the special case of segment interchanging where the segments to be interchanged are of the same size.

In this paper we study the performance of the segment-interchanging methods on a computer with a hierarchical memory. Especially, we consider the largest memory level that is too small to fit the whole input array. We call that particular memory level simply the **cache**. In our analytical results we assume that the



cache is **fully associative** (for the definition, see, for instance, [8, Section 5.2]), and that replacement of blocks is accomplished by the **least-recently-used** (LRU) strategy. Throughout this paper we use  $M$  to denote the **capacity** of the **cache** and  $B$  the **capacity** of a **block**, both measured in elements. To assess the quality of the methods, two measures have been used in earlier research papers: the number of **element moves** and the number of **element exchanges**. In our analysis we count both the number of **memory references**, as recommended by Knuth in his recent book [9, pp. 464–465], and the number of **block transfers** between the cache and the next larger memory level, in a very similar manner as proposed by Aggarwal and Vitter [1].

## 2 Known Methods

Basically, there exists four different approaches for solving the problem, even though in different sources the implementation details can vary. The four methods are: the copying method (see, e.g., [13]), the successive-segment-swapping method [7] (see also [6, Section 18.1]), the reversal method (see, e.g., [11, Exercise 5.5-3] or [7,12]), and the dolphin method [5] (see also [10, Exercise 1.3.3-34]) as it was named by Ted Nelson (as cited in [7]). The first of these methods needs extra space, whereas the others operate in-place. A detailed discussion of the methods can be found in [7], and C++ realizations of the in-place methods are available in the Silicon Graphics Inc. implementation of the STL (SGI STL, version 3.2) [14].

**Copying Method.** In the copying method the shorter of the given segments is copied to an auxiliary array, the longer of the segments is moved to its destination, and finally the saved copy is moved from the auxiliary array to its destination. The main drawback of this method is that an extra array of size  $\min\{m, n\}$  is required for its operation. To calculate the size of a segment in constant time, random-access iterators must be provided, but even if the sizes of the segments cannot be calculated, it is still possible to implement the method by using bidirectional iterators. We leave the details for an interested reader.

As the experiments reported by Shene [13] show, the in-place methods have difficulties in beating this simple method. This was the case even if Shene used a naive implementation, where the second segment was always moved to the auxiliary array.

**Successive-Segment-Swapping Method.** Let  $\alpha$  and  $\beta$  be the two segments to be interchanged, and assume that  $\alpha$  is the shorter of the two. The input may be either  $\alpha\beta$ , in which case the return value should be  $\beta\alpha$ , or it may be  $\beta\alpha$ , in which case  $\alpha\beta$  should be returned. Further, let  $\beta'$  be a prefix of  $\beta$  that is as long as  $\alpha$ , and let  $\beta = \beta'\beta''$ . Now the equal-sized segments  $\alpha$  and  $\beta'$  are swapped, and the reduced problem — either  $\alpha\beta''$  if  $\alpha$  was the first segment, or  $\beta''\beta'$  otherwise — is solved recursively. The key observation is that two equal-sized segments can be interchanged in-place simply by exchanging the elements one by one. Also, the tail recursion can be easily removed so no recursion stack will be necessary. This approach for the segment-interchanging problem was proposed by Gries

**Table 1.** Performance of the known segment-interchanging methods.

Method	Element Exchanges	Element Moves	Memory References	Block Transfers
Copying	—	$(3/2)N$	$3N$	$3N/B + O(1)$
Swapping	$N - \gcd(m, n)$	$3(N - \gcd(m, n))$	$4(N - \gcd(m, n))$	$4N/B + 4N/M$
Reversal	$N$	$3N$	$4N$	$4N/B + O(1)$
Dolphin	—	$N + \gcd(m, n)$	$2N$	$2N$

and Mills in a technical report [7], but it appears also in the book by Gries [6, Section 18.1]. In the SGI STL there is a particularly elegant implementation of the method relying only on forward iterators.

**Reversal Method.** Let  $\alpha^R$  denote the reversal of segment  $\alpha$ . In the reversal method the interchange is accomplished with three successive reversals, first forming  $\alpha^R\beta$ , then  $\alpha^R\beta^R$ , and finally  $(\alpha^R\beta^R)^R$ , which is  $\beta\alpha$ . Since the reversal of a segment is easily accomplished in-place, segment interchanging can also be carried out in-place. As pointed out both in [6, p. 302] and in [12], this method is part of the folklore and its origin is unknown. The reversal method is implemented in the SGI STL using bidirectional iterators.

**Dolphin Method.** The dolphin algorithm is based on a group theoretic argument. The permutation corresponding to the segment interchange has  $d = \gcd(m, n)$  disjoint cycles, each of length  $N/d$ . Furthermore, the index set of the elements that belong to one cycle is  $\{i_0, i_0 + d, i_0 + 2d, \dots, i_0 + N - d\}$ , so each element in  $\{0, \dots, \gcd(m, n) - 1\}$  belongs to a separate cycle. In the dolphin method the cycles are processed one at a time.

**Performance Summary.** In Table 1 the performance of the known segment-interchanging methods is summarized. The upper bounds for the number of element moves and element exchanges are taken from the papers [7,13], and those for the number of memory references and block transfers are derived in the full version of this paper. It is known that for any segment-interchanging method  $N - \gcd(m, n)$  is a lower bound for the number of element exchanges [12], and  $N + \gcd(m, n)$  that for the number of element moves [3,13]. Since each element has to be read at least once and written at least once,  $2N$  is a trivial lower bound for the number of memory references. In the best case all these memory references could be carried out blockwise, so  $2N/B$  is a lower bound for the number of block transfers.

### 3 Reengineered Methods

In this section we present a straightforward improvement of the reversal method and a new method, based on the dolphin method, which repairs its bad cache behaviour with only a minor extra cost in terms of memory references.

```

1 Rotate( $\mathbf{A}[0, \dots, m, \dots, N]$ ) :
2    $i, j, k, l := 0, m - 1, m, N - 1$ 
3   while ( $i < j \wedge k < l$ )
4      $\mathbf{A}[i], \mathbf{A}[j], \mathbf{A}[k], \mathbf{A}[l] := \mathbf{A}[k], \mathbf{A}[i], \mathbf{A}[l], \mathbf{A}[j]$ 
5      $i, j, k, l := i + 1, j - 1, k + 1, l - 1$ 
6   comment: Only one of the next two loops will be executed.
7   while ( $k < l$ )
8      $\mathbf{A}[i], \mathbf{A}[k], \mathbf{A}[l] := \mathbf{A}[k], \mathbf{A}[l], \mathbf{A}[i]$ 
9      $i, k, l := i + 1, k + 1, l - 1$ 
10  while ( $i < j$ )
11     $\mathbf{A}[i], \mathbf{A}[j], \mathbf{A}[l] := \mathbf{A}[l], \mathbf{A}[i], \mathbf{A}[j]$ 
12     $i, j, l := i + 1, j - 1, l - 1$ 
13  while ( $i < l$ )
14     $\mathbf{A}[i], \mathbf{A}[l] := \mathbf{A}[l], \mathbf{A}[i]$ 
15     $i, l := i + 1, l - 1$ 

```

**Fig. 1.** The tuned reversal method.

**Segment Assignment.** We extend the parallel assignment to handle an assignment of array segments. Let  $\mathbf{A}$  be an array and  $s_1, s_2, t_1, t_2$  be indices such that  $s_1 < s_2$  and  $t_1 < t_2$ . We write  $\mathbf{A}[s_1, \dots, s_2] := \mathbf{A}[t_1, \dots, t_2]$  as a shorthand for  $\mathbf{A}[s_1], \dots, \mathbf{A}[s_2 - 1] := \mathbf{A}[t_1], \dots, \mathbf{A}[t_2 - 1]$ . If the two segments are not of equal length, the length of the shorter segment determines how many elements are assigned. In particular, we allow one of the two segments to be of unspecified length.

### 3.1 Reversal Method Revisited

A simple optimization of the reversal method is possible by fusing the three reversal loops into one. Recall that the sizes of two non-intersecting segments of array  $\mathbf{A}$  being interchanged are  $m$  and  $n$ , respectively. The first iteration of the fused loop will handle the four elements  $\mathbf{A}[0]$ ,  $\mathbf{A}[m - 1]$ ,  $\mathbf{A}[m]$ , and  $\mathbf{A}[N - 1]$ . The unoptimized method would execute three element exchanges:  $\mathbf{A}[0], \mathbf{A}[m - 1] := \mathbf{A}[m - 1], \mathbf{A}[0]$ ;  $\mathbf{A}[m], \mathbf{A}[N - 1] := \mathbf{A}[N - 1], \mathbf{A}[m]$ ;  $\mathbf{A}[0], \mathbf{A}[N - 1] := \mathbf{A}[N - 1], \mathbf{A}[0]$ . But because these exchanges are now brought together, they can be optimized to a single parallel assignment:  $\mathbf{A}[0], \mathbf{A}[m - 1], \mathbf{A}[m], \mathbf{A}[N - 1] := \mathbf{A}[m], \mathbf{A}[0], \mathbf{A}[N - 1], \mathbf{A}[m - 1]$ , which is equivalent to five sequential assignments. If  $s = \min\{m, n\}$  and  $S = \max\{m, n\}$ , this process can be repeated  $\lfloor s/2 \rfloor$  times after which a similar kind of permutation of three elements must be carried out  $\lfloor S/2 \rfloor - \lfloor s/2 \rfloor$  times. The remaining  $\lfloor N/2 \rfloor - \lfloor S/2 \rfloor$  iterations will do ordinary element exchanges. Pseudo-code for the tuned reversal method is given in Figure 1.

This optimized method will bring  $N/2$  elements directly to their final destinations, so these elements are read and written only once. The other half of the elements are still to be read and written twice. Hence, the number of memory references reduces to  $3N$ . Even the optimized method traverses the array

sequentially, visiting half of the elements twice, so the number of block transfers is bounded by  $3N/B + O(1)$ . Clearly, this bound is reachable if  $M \leq \min\{m, n\}$ .

Both variants of the reversal method scan the memory sequentially so they have good spatial locality, but in their practical behaviour there are some differences. In the standard implementation two iterators move simultaneously through memory in opposite directions, so a direct-mapped cache is enough to avoid most interference. In the first part of the tuned version, four iterators are maintained simultaneously, moving pairwise in opposite directions in memory, which requires more registers and at least a 2-way set-associative cache to guarantee good performance.

### 3.2 Dolphin Method Revisited

In this section we describe a new segment-interchanging method which is an adaptation of the dolphin method. It behaves similarly to the copying method when available buffer space is larger than the smaller segment. The basic idea is to move nearby elements simultaneously without giving up the optimality of the dolphin method. That is, most elements are moved directly to their final destinations, but this is mostly done in a blockwise manner so that spatial locality is achieved.

As earlier, we let  $m$  and  $n$  denote the respective sizes of the segments to be interchanged, and assume for simplicity that  $m > n$ . Further, we let  $M'$  denote the size of the **buffer** used. The actual size depends on the size of the cache,  $M$ . Suppose now that two neighbouring elements stored in  $\mathbf{A}[k]$  and  $\mathbf{A}[k+1]$  have been moved to the buffer. The empty slots are to be filled with the elements from the positions  $(k+m) \bmod N$  and  $(k+1+m) \bmod N$ , respectively. Unfortunately, we cannot be certain that these two elements are adjacent in memory. The problem that must be solved is: how to ensure that the chunk of empty slots which moves across the array remains contiguous, and how to keep track of the positions that have already been processed. Furthermore, this book-keeping must be done implicitly using only a constant amount of extra space.

**Definitions.** To begin with, we introduce some notation. Indices to array  $\mathbf{A}$  are regarded as elements in the Euclidean ring  $\mathbb{Z}_N$  with modular arithmetic. For indices  $\bar{a}, \bar{b} \in \mathbb{Z}_N$ , we say that  $\bar{a} < \bar{b}$  if and only if  $a < b$ , for the representatives  $a, b \in \{0, \dots, N-1\}$ . The **index translation function**  $T_m : \mathbb{Z}_N \rightarrow \mathbb{Z}_N$ , or in brief  $T$ , is defined by  $T(a) = a + m$ . An **index pattern**  $P$  is any sequence of indices resulting from repeated application of  $T$  to 0. The length of  $P$ , denoted  $|P|$ , is the number of indices in  $P$ . The last index in  $P$  is  $\text{last}(P) = T^{(|P|)}(0)$ . Note that the first index in a pattern is  $m$ , not 0. We use the notation  $\min P$  and  $\max P$  to denote the smallest respectively the largest of the indices in pattern  $P$ . Patterns are not allowed to be longer than  $N/\gcd(m, n)$ .

Consider the intermediate numbers generated by Euclid's algorithm for calculating the greatest common divisor,  $\gcd(m, n)$ . Let  $r_1 = m$ ,  $r_2 = n$ , and  $r_i = r_{i-2} \bmod r_{i-1}$  for  $i > 2$ . (Note: we still assume that  $m > n$ .) This defines a finite sequence, called here the **remainder sequence**, where the last non-zero element is  $\gcd(m, n)$ . The **quotient sequence** associated with the remainder

```

1 Rotate( $\mathbf{A}[0, \dots, m, \dots, N]$ ) :
2   assert:  $r_j \neq 0 \wedge (j \text{ is even} \Leftrightarrow m < N - m)$ 
3    $\text{hole} := 0$ ;  $\text{buffer}[0, \dots, r_j] := \mathbf{A}[0, \dots, r_j]$ 
4   for  $i := 1$  to  $L_{j+1} - 1$ 
5     assert:  $r_j \leq T(\text{hole}) \leq N - r_{j+1} - r_j$ 
6      $\mathbf{A}[\text{hole}, \dots, \text{hole} + r_j] := \mathbf{A}[T(\text{hole}), \dots, T(\text{hole}) + r_j]$ 
7      $\text{hole} := T(\text{hole})$ 
8   assert:  $T(\text{hole}) = N - r_{j+1}$ 
9    $\mathbf{A}[\text{hole} + r_{j+1}, \dots] := \text{buffer}[0, \dots, r_j - r_{j+1}]$ 
10  if ( $r_{j+1} \neq 0$ )
11    for  $i := 1$  to  $L_j$ 
12      assert:  $r_j + r_{j+1} \leq T(\text{hole}) + r_{j+1} \leq N - r_{j-1}$ , except when  $i = 1$ 
13       $\mathbf{A}[\text{hole}, \dots, \text{hole} + r_{j+1}] := \mathbf{A}[T(\text{hole}), \dots, T(\text{hole}) + r_{j+1}]$ 
14       $\text{hole} := T(\text{hole})$ 
15    assert:  $T(\text{hole}) + r_{j+1} = r_j$ 
16     $\mathbf{A}[\text{hole}, \dots] := \text{buffer}[r_j - r_{j+1}, \dots, r_j]$ 

```

**Fig. 2.** The load-once algorithm.

sequence is defined by  $q_i = \lfloor r_{i-2}/r_{i-1} \rfloor$  for  $i > 2$ . With these definitions we have that  $r_{i-2} = q_i r_{i-1} + r_i$  for  $i > 2$ . We also define the **length sequence** associated with the remainder sequence and the corresponding quotient sequence by  $L_1 = L_2 = 1$  and  $L_i = q_i L_{i-1} + L_{i-2}$  for  $i > 2$ . Note that  $N = L_{j+1} r_j + L_j r_{j+1}$ , for any  $j > 0$ , where  $r_j \neq 0$ .

**Load Once Case.** As the first step we will construct an algorithm which only loads elements into the buffer area once. For that purpose we will need the following theorem which specifies the order in which the positions in the vicinity of 0 are visited when  $T$  is applied repeatedly to 0. This theorem will allow the selection of an appropriate buffer size  $M'$  in many cases, and it provides the foundation for our load-once algorithm.

**Theorem 1.** *Assume that  $m > n$  and let  $P$  be an index pattern of length  $|P| = L_j$  for some  $j > 1$ , where  $r_j \neq 0$ . Then if  $j$  is even,  $\text{last}(P) = \max P = -r_j$  and  $\min P = r_{j-1}$ ; and if  $j$  is odd,  $\max P = -r_{j-1}$  and  $\text{last}(P) = \min P = r_j$ . Furthermore, if  $m < n$ , swap the words 'even' and 'odd' in the above.*

**Proof:** The proof is by induction on  $j$ . Details are given in the full version of the paper.  $\square$

By Theorem 1 and the identity  $N = L_{j+1} r_j + L_j r_{j+1}$  we can write a load-once algorithm as given in Figure 2. The algorithm can use any non-zero number  $r_j$  from the remainder sequence as the buffer size. To prove the correctness of the algorithm we must justify that no position is processed twice, and that exactly  $N$  positions are processed. The latter is seen directly by the identity  $N = L_{j+1} r_j + L_j r_{j+1}$ , while the former is a result of Theorem 1. The assertions in lines 8 and 15 follow directly from Theorem 1, and it is obvious that the overlap of the source segment with the contents of the buffer taken from  $[0, \dots, r_j]$  is handled correctly. What remains is to justify the correctness of the assertions in lines 5 and 12, which ensures that no other overlaps with  $[0, \dots, r_j]$

```

1 Rotate( $\mathbf{A}[0, \dots, m, \dots, N]$ ) :
2   assert:  $r_j \neq 0 \wedge (j \text{ is even} \Leftrightarrow m < N - m)$ 
3    $hole := 0$ ;  $buffer[0, \dots, r_j] := \mathbf{A}[0, \dots, r_j]$ 
4   for  $i := 1$  to  $q_{j+1}$ 
5     assert:  $hole = (i - 1)r_j$ 
6     for  $k := 1$  to  $L_j$  {  $SegmentMove(r_j)$ ;  $hole := T(hole)$  }
7   assert:  $hole = q_{j+1}r_j = r_{j-1} - r_{j+1}$ 
8   for  $i := 1$  to  $L_{j-1} - 1$  {  $SegmentMove(r_j)$ ;  $hole := T(hole)$  }
9   assert:  $T(hole) = N - r_{j+1}$ 

```

**Fig. 3.** First part of the intermediate load-once algorithm.

occurs. Consider the assertion in line 5. The elements to fill the hole are to be taken from the sequence of indices  $[T(hole), \dots, T(hole) + r_j]$ . The assertion  $r_j \leq T(hole)$  is a direct consequence of Theorem 1 and so it is valid, but the upper bound on  $T(hole)$  provided by Theorem 1 is only  $T(hole) < -r_{j+1}$ . Anyway, the bound can be strengthened to  $T(hole) \leq -r_{j+1} - r_j$  by the following argument: Suppose that  $T(hole) + r_j > -r_{j+1}$  at iteration  $0 < t < L_{j+1}$ . Then we have  $T^{(L_{j+1})}(0) = -r_{j+1} \in T^{(t)}([0, \dots, r_j])$ . But  $T$  is a bijection and hence  $T^{(L_{j+1}-t)}(0) \in [0, \dots, r_j]$ , or in other words  $T(hole) < r_j$  in iteration  $L_{j+1} - t > 0$ , which is in contradiction with the already established bound  $r_j \leq T(hole)$ . The lower bound on  $T(hole)$  in line 12 is established with a similar argument.

The problem with the algorithm in Figure 2 is that, given the maximum buffer size  $M$ , we cannot be certain that there exists  $r_j$  such that  $0 < r_j \leq M$ ; and even if such a value exists, it may be that the candidate  $r_j$  is too small to allow efficient buffering. In fact, it could happen that the only candidate is 1, in which case the load-once algorithm has the same memory access pattern as the original dolphin algorithm. To remedy this problem we must somehow allow the use of a buffer whose size is larger than  $r_j$ .

**Multiple Load Case.** As an intermediate step we use the relation  $L_{j+1} = q_{j+1}L_j + L_{j-1}$  to split the **for**-loop in Figure 2, line 4 into a pair of nested loops and an extra loop. The resulting loops are presented in Figure 3. To improve readability, the rather uninteresting assignments  $\mathbf{A}[hole, \dots, hole + size] := \mathbf{A}[T(hole), \dots]$  have been exchanged with the place-holder  $SegmentMove(size)$ . Since this intermediate algorithm is a trivial modification of the algorithm in Figure 2, their functioning is identical, but the interesting attribute of the intermediate algorithm is the assertions in lines 5 and 7 which with some consideration will allow reloading of the buffer in a controlled fashion.

Let us take a closer look at the loop in Figure 3, line 4. The assertion in line 5 tells us that, if we start with a buffer size  $M' > r_j$ , we will have to unload  $M' - r_j$  elements after  $L_j - 1$  segment moves. We have room for  $\lfloor M'/r_j \rfloor$  segments of size  $r_j$  in the buffer simultaneously and in line 5 the buffer already holds  $r_j$  elements. Hence, if we fill the buffer in the top of the loop with  $\lfloor M'/r_j \rfloor r_j - r_j$  extra elements, i.e.  $M' = \lfloor M'/r_j \rfloor r_j$ , we must reduce the number of iterations from  $q_{j+1}$  to  $\lfloor q_{j+1}r_j/M' \rfloor$  which leaves  $q_{j+1} \bmod (M'/r_j)$  iterations to be handled

```

1 Rotate( $\mathbf{A}[0, \dots, m, \dots, N]$ ) :
2   assert:  $r_j \neq 0 \wedge (j \text{ is even} \Leftrightarrow m < N - m)$ 
3    $M' = \lfloor M/r_j \rfloor r_j$ 
4    $hole := 0$ ;  $buffer[0, \dots, r_j] := \mathbf{A}[0, \dots, r_j]$ 
5   for  $i := 1$  to  $\lfloor q_{j+1}r_j/M' \rfloor$ 
6     assert:  $hole = (i - 1)M'$ 
7      $extra\_buffer[0, \dots, M' - r_j] := \mathbf{A}[hole + r_j, \dots]$ 
8     for  $k := 1$  to  $L_j - 1$  {  $SegmentMove(M')$ ;  $hole := T(hole)$  }
9     assert:  $T(hole) = (i - 1)M' + r_j$ 
10     $\mathbf{A}[hole, \dots] := extra\_buffer[0, \dots, M' - r_j]$ 
11     $hole := hole + M' - r_j$ 
12     $SegmentMove(r_j)$ ;  $hole := T(hole)$ 
13  assert:  $hole = \lfloor q_{j+1}r_j/M' \rfloor M'$ 
14  for  $i := 1$  to  $q_{j+1} \bmod (M'/r_j)$ 
15    assert:  $hole = \lfloor q_{j+1}r_j/M' \rfloor M' + (i - 1)r_j$ 
16    for  $i := 1$  to  $L_j$  {  $SegmentMove(r_j)$ ;  $hole := T(hole)$  }
17  assert:  $hole = q_{j+1}r_j = r_{j-1} - r_{j+1}$ 

```

**Fig. 4.** Blockwise implementation of the loop in Figure 3 line 4.

separately. An implementation of this idea is shown in Figure 4. For readability we use a separate buffer area for the extra loads, but it should be emphasized that there is never loaded more than  $M$  elements into the two buffers simultaneously, so the combined buffers still fit in the cache.

We could stop here and use the algorithm we have developed so far, with only a few modifications to handle the case when  $r_j = 0$ , but further opportunity for refilling the buffer exists in the last part of the algorithm. However, the principle behind this tuning is simply ‘more of the same’ so we will omit the details. To handle the situation when  $r_j = 0$ , which leaves  $M'$  and  $q_{j+1}$  undefined, we let  $M' = M$  when  $r_j = 0$ . Furthermore, we have the identity  $\lfloor r_{j-1}/M' \rfloor = \lfloor q_{j+1}r_j/M' \rfloor$  when  $r_j \neq 0$ , that allows the removal of all occurrences of  $q_{j+1}$  from the algorithm altogether. The multiple-load algorithm is presented in Figure 5.

The complete algorithm, named **blockwise dolphin**, consists of two symmetric copies of the multiple-load algorithm to handle the cases when  $(j \text{ is even} \Leftrightarrow m < N - m)$  is true respectively false.

**Performance Analysis.** Each element is either moved directly to its final destination, causing two memory references per element, or moved via the buffer area, causing four memory references per element, so the total number of memory references executed is bounded by  $4N$ . Indeed, if  $\min\{m, n\} \leq M'$ , the number of elements moved via the buffer is  $N(M' - \min\{m, n\})/M'$  which is close to  $N$ , if  $\min\{m, n\}$  is small, so it seems that the blockwise dolphin is no better than the reversal method or the successive-segment-swapping method. However, as is shown in the full version of the paper, the algorithm will incur at most  $= \frac{c}{c-1}(2N/B + 6N/M') + O(1)$  block transfers for a constant  $c > 1$  such that  $M/2 < cM' < M$ .

```

1 Rotate(A[0, ..., m, ..., N]) :
2   assert:  $r_j \leq M \wedge r_{j-1} \neq 0 \wedge (j \text{ is even} \Leftrightarrow m < N - m)$ 
3   if ( $r_j \neq 0$ )
4      $M' := \lfloor M/r_j \rfloor r_j$ 
5   else
6      $M' := M$ 
7    $M'' := r_{j-1} \bmod M'$ 
8    $hole := 0$ ;  $buffer[0, \dots, r_j] := \mathbf{A}[0, \dots, r_j]$ 
9   for  $i := 1$  to  $\lfloor r_{j-1}/M' \rfloor$ 
10    assert:  $hole = (i - 1)M'$ 
11     $extra\_buffer[0, \dots, M' - r_j] := \mathbf{A}[hole + r_j, \dots]$ 
12    for  $k := 1$  to  $L_j - 1$  {  $SegmentMove(M')$ ;  $hole := T(hole)$  }
13    assert:  $T(hole) = (i - 1)M' + r_j$ 
14     $\mathbf{A}[hole, \dots] := extra\_buffer[0, \dots, M' - r_j]$ 
15     $hole := hole + M' - r_j$ 
16     $SegmentMove(r_j)$ ;  $hole := T(hole)$ 
17  assert:  $hole = \lfloor r_{j-1}/M' \rfloor M' = r_{j-1} - M''$ 
18   $extra\_buffer[0, \dots, M'' - r_{j+1}] := \mathbf{A}[hole + r_j, \dots]$ 
19  for  $i := 1$  to  $L_{j-1} - 1$  {  $SegmentMove(M'' + r_j - r_{j+1})$ ;  $hole := T(hole)$  }
20  assert:  $T(hole) = N - M''$ 
21   $\mathbf{A}[hole + r_{j+1}, \dots] := buffer[0, \dots, r_j - r_{j+1}]$ 
22  for  $i := 1$  to  $L_j - L_{j-1}$  {  $SegmentMove(M'')$ ;  $hole := T(hole)$  }
23  assert:  $T(hole) = r_{j-1} - M'' + r_j$ 
24   $\mathbf{A}[hole, \dots] := extra\_buffer[0, \dots, M'' - r_{j+1}]$ 
25   $hole := hole + M'' - r_{j+1}$ ;
26  if ( $r_{j+1} \neq 0$ )
27    for  $i := 1$  to  $L_{j-1}$  {  $SegmentMove(r_{j+1})$ ;  $hole := T(hole)$  }
28  assert:  $T(hole) = r_j - r_{j+1}$ 
29   $\mathbf{A}[hole, \dots] := buffer[r_j - r_{j+1}, \dots, r_j]$ 

```

Fig. 5. The multiple-load algorithm.

We see that the factor  $\frac{c}{c-1}$  suggests the use of a small buffer, while the term  $6N/M'$  suggests that a large buffer should be used. Anyway, typical cache configurations are large enough to allow a reasonable choice to be made. As an example consider a cache configuration with  $M = 16384$  and  $B = 16$ . If we, somewhat arbitrarily, select  $c = 8$ , so that  $M/16 < M' \leq M/8$ , we get that the number of block transfers is bound by  $(8/7)(2N/B + 6N/(M/8)) + O(1) = 2.3N/B + O(1)$ .

## 4 Implementation Details and Experimental Results

To validate the theoretical findings, we carried out a collection of experiments with the programs available in the SGI STL and our tuned programs. The SGI STL provides an overloaded function template:

```
void rotate(iterator first, iterator middle, iterator beyond)
```

which implements the three in-place methods described in Section 2. The methods are provided as separate routines for random-access iterators, bidirectional



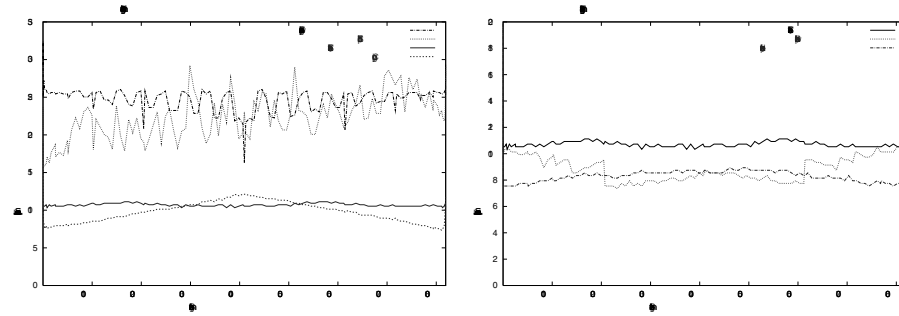
iterators, and forward iterators. The proper routine is selected automatically, based on the capability of the parameters given, by the overload resolution mechanism of C++. Our two tuned methods were implemented to be plug-in replacements for the SGI STL random-access iterator implementation. The buffer used by the blockwise dolphin was restricted to a size no larger than  $M/8$ . For comparison we also implemented the copying method. In our implementation the STL functions `get_temporary_buffer()` and `return_temporary_buffer()` are used to allocate and deallocate uninitialized memory.

The experiments were repeated in several different computers with a hierarchical memory. The results were similar in all the computers we tried, so we report here only those obtained on a Compaq AlphaServer DS20. The characteristics of this computer are as follows. It has two 500 MHz Alpha CPUs (21264)—only one is used during our experiments—64 kbyte on-chip 2-way set-associative cache with a line-size of 64 bytes, and a direct-mapped 4 Mbyte off-chip cache, also with a cache line-size of 64 bytes. The experiments were carried out on dedicated hardware, so no cache interference from other sources was expected.

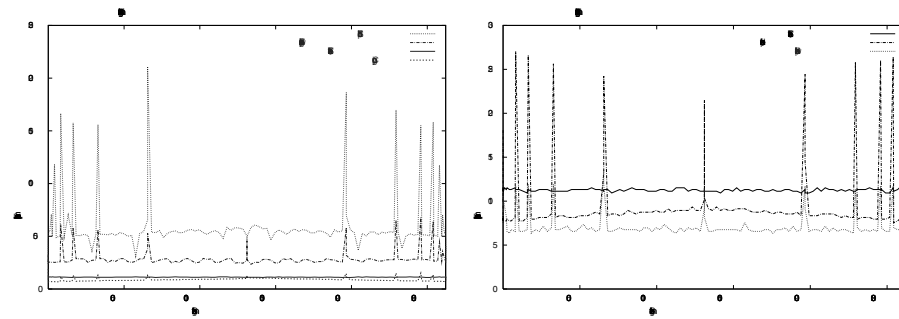
The programs were compiled by using the Digital C++ V6.1-027 compiler provided by the hardware manufacturer. The object codes were generated with the following command line options `-fast -arch ev6 -tune ev6 -unroll 8` controlling compiler optimizations, and the options `-newcxx -std ansi` specifying the variant of the programming language.

In all of our experiments, input was provided to the SGI STL in the form of pointers to an array. Since pointers are random-access iterators, the SGI STL would normally select the dolphin routine for such an input, but we forced the SGI STL to choose the other routines by by-passing the standard call-interface. As input data we used 4-byte integers. In our experiments the sizes of the input were: 8 192 (Figure 6), 262 144 (Figure 7), and 8 388 608 (Figure 8). The smallest problems fitted easily in the first-level cache, whereas the medium-size problems were too large to fit there, but they fitted in the second-level cache, and finally the largest problems were too large for the second-level cache, but small enough to fit in the main memory. For each of the three problem sizes, two sets of rotation values,  $m$ , were used. In the first set the numbers were designed specifically to make  $\gcd(m, n)$  large by selecting  $m$  to be a power of 2, while in the other set  $m$  was chosen to be the multiples of a prime close to  $N/100$ , thereby causing  $\gcd(m, n)$  to be small. Each experiment was repeated a large number of times to increase the timing precision. To ensure that no part of the input array was already in the cache when timing was started, no two trials used the same memory area.

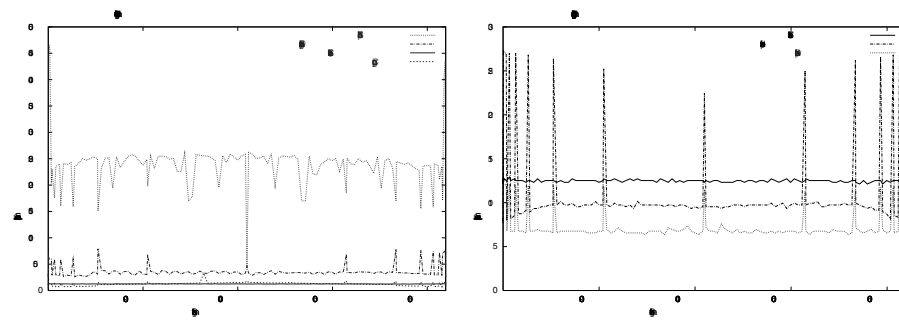
In Figures 6, 7, and 8 two graphs are displayed. The graphs on the left compare the SGI STL programs and the copying program, while the graphs on the right compare the new programs to the reversal program, which was the best among the SGI STL programs. We observe that the dolphin program is about 20 (twenty) times slower than the reversal program, when the problem does not fit in the second-level cache. According to the graphs on the right, the blockwise dolphin achieved almost a factor of 2 speed-up compared to the



**Fig. 6.** Execution time per element for the STL and our tuned programs on a Compaq AlphaServer DS20 when the input fits in the first-level cache.



**Fig. 7.** Execution time per element for the STL and our tuned programs on a Compaq AlphaServer DS20 when the input fits in the second-level cache.



**Fig. 8.** Execution time per element for the STL and our tuned programs on a Compaq AlphaServer DS20 when the input does not fit in any of the caches.

reversal program on the medium-sized and large inputs, while the difference was not so big on the small inputs. Especially, the blockwise dolphin program uses less time per element on the large inputs than on the small inputs, suggesting that there is a non-trivial overhead for small inputs. To confirm this we ran the reversal program, the tuned reversal program, and the blockwise dolphin program with  $N = 4, m = 2$ . The result was that the tuned reversal program was 1.6 times faster than the reversal program which in turn was 5.3 times faster than the blockwise dolphin program.

## References

1. A. AGGARWAL AND J. S. VITTER, The input/output complexity of sorting and related problems, *Communications of the ACM* **31** (1988), 1116–1127.
2. J. BENTLEY, *Programming Pearls*, 2nd Edition, Addison-Wesley, Reading (2000).
3. K. DUDZIŃSKI AND A. DYDEK, On stable minimum storage merging algorithm, *Information Processing Letters* **12** (1981), 5–8.
4. S. DVOŘÁK AND B. ĐURIAN, Merging by decomposition revisited, *The Computer Journal* **31** (1988), 553–556.
5. W. FLETCHER AND R. SILVER, Interchange of two blocks of data, *Communications of the ACM* **9** (1966), 326.
6. D. GRIES, *The Science of Programming*, Springer-Verlag, New York (1981).
7. D. GRIES AND H. MILLS, *Swapping sections*, Technical report, Department of Computer Science, Cornell University, Ithaca, New York (1981).
8. J. L. HENNESSY AND D. A. PATTERSON, *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, Inc., San Francisco (1996).
9. D. E. KNUTH, *The Stanford GraphBase: A Platform for Combinatorial Computing*, Addison-Wesley, Reading (1993).
10. D. E. KNUTH, *Fundamental Algorithms, The Art of Computer Programming* **1**, 3rd Edition, Addison-Wesley, Reading (1997).
11. D. E. KNUTH, *Sorting and Searching, The Art of Computer Programming* **3**, 2nd Edition, Addison-Wesley, Reading (1998).
12. J. L. MOHAMMED AND C. S. SUBI, An improved block-interchange algorithm, *Journal of Algorithms* **8** (1987), 113–121.
13. C. K. SHENE, An analysis of two in-place array rotation algorithms, *The Computer Journal* **40** (1997), 541–546.
14. SILICON GRAPHICS COMPUTER SYSTEMS, INC., *Standard Template Library Programmer's Guide*, Worldwide Web Document (1999). Available at <http://www.sgi.com/Technology/STL/>.

# Two-Dimensional Arrangements in CGAL and Adaptive Point Location for Parametric Curves<sup>\*</sup>

Iddo Hanniel and Dan Halperin

Department of Computer Science  
Tel Aviv University, Tel Aviv 69978, Israel  
{hanniel,halperin}@math.tau.ac.il

**Abstract.** Given a collection  $\mathcal{C}$  of curves in the plane, the *arrangement* of  $\mathcal{C}$  is the subdivision of the plane into vertices, edges and faces induced by the curves in  $\mathcal{C}$ . Constructing arrangements of curves in the plane is a basic problem in computational geometry. Applications relying on arrangements arise in fields such as robotics, computer vision and computer graphics. Many algorithms for constructing and maintaining arrangements under various conditions have been published in papers. However, there are not many implementations of (general) arrangements packages available. We present an implementation of a generic and robust package for arrangements of curves that is part of the CGAL<sup>1</sup> library. We also present an application based on this package for adaptive point location in arrangements of parametric curves.

## 1 Introduction

This paper is concerned with a fundamental structure in Computational Geometry – an *arrangement* of curves in the plane. Given a collection  $\mathcal{C}$  of curves in the plane, the arrangement of  $\mathcal{C}$  is the subdivision of the plane into vertices, edges and faces induced by the curves in  $\mathcal{C}$ . Arrangements of lines in the plane, as well as arrangements of hyperplanes in higher dimensions have been extensively studied in computational geometry [7], with applications to a variety of problems. Arrangements of general curves have also been studied producing combinatorial results (e.g., the zone theorem) and algorithmic ones (e.g., algorithms for constructing substructures such as the lower envelope) [15,21]. These algorithms can be applied to many problems in several “physical world” application domains. For example, the problem of motion planning of a robot with two degrees of freedom (under reasonable assumptions that the shape of the robot and of the obstacles consist of algebraic arcs of some constant degree), can be solved using an arrangement of algebraic curves.

---

<sup>\*</sup> This work has been supported in part by ESPRIT IV LTR Projects No. 21957 (CGAL) and No. 28155 (GALIA), by The Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (Center for Geometric Computing and its Applications), by a Franco-Israeli research grant (monitored by AFIRST/France and The Israeli Ministry of Science), and by the Hermann Minkowski – Minerva Center for Geometry at Tel Aviv University. Dan Halperin has been also supported in part by the USA-Israel Binational Science Foundation.

<sup>1</sup> <http://www.cs.uu.nl/CGAL/>

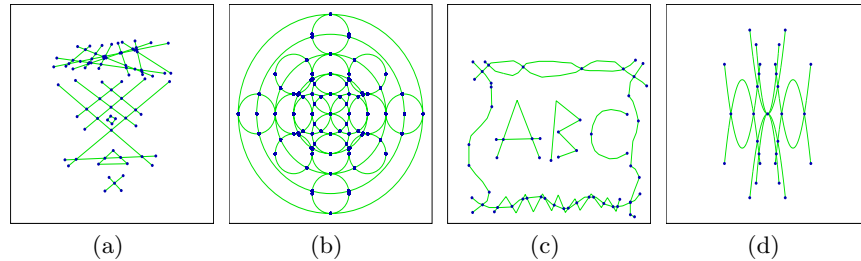
There are not many implementations of arrangements reported in the literature. The Algorithm Design Manual [23] records only two implementations: *arrange* [13] – a package for maintaining arrangements of polygons, and LEDA<sup>2</sup> – the Library of Efficient Data-structures and Algorithms [18,19] – which provides efficient algorithms (sweep-line and randomized-incremental) to construct the planar map induced by a set of (possibly intersecting) segments. LEDA also supports a subdivision class for planar maps of non-intersecting segments with point location queries. Both implementations are restricted to dealing with linear objects – line segments or polygons. Amenta [1,2] also points to the *pploc* package for answering vertical ray shooting queries in a set of non-intersecting line segments (which seems much more restricted than the other packages). The MAPC library [17] is a software package for manipulating algebraic points and curves. It has been used to construct the combinatorial structure of an arrangement of algebraic curves. However, it is an algebraic package rather than an arrangement package. Neagu and Lacolle [20] provide an algorithm and an implementation that takes as input a set of Bézier curves and outputs a set of polygonal lines whose arrangement has the same combinatorial structure. Their implementation does not build the arrangement data structure. We make use of some of their theoretical results in our application.

In this paper we present the design and implementation of a generic and robust software package for representing and manipulating arrangements of general curves. The package is part of CGAL – the Computational Geometry Algorithms Library, which is a collaborative effort of several academic institutes in Europe [5,8], to develop a C++ software library of geometric data structures and algorithms. The library’s main design goals are robustness, genericity, flexibility and efficiency. To achieve these CGAL adopted the *generic programming paradigm* (see [3,4]). These goals (particularly the first three) stood before us in our implementation as well.

We now give formal definitions of planar maps and arrangements. A *planar map* is a planar embedding of a planar graph  $G$  such that each arc of  $G$  is embedded as a bounded curve, the image of a node of  $G$  is a *vertex*, and the image of an arc is an *edge*. We require that each edge be a bounded  $x$ -monotone curve<sup>3</sup>. A *face* of the planar map is a maximal connected region of the plane that does not contain any vertex or edge. Given a finite collection  $\mathcal{C}$  of (possibly intersecting and not necessarily  $x$ -monotone) curves in the plane, we construct a collection  $\mathcal{C}''$  of curves in two steps: First we decompose each curve in  $\mathcal{C}$  into maximal  $x$ -monotone curves, thus obtaining a collection  $\mathcal{C}'$ . We then decompose each curve in  $\mathcal{C}'$  into maximal connected pieces not intersecting any other curve in  $\mathcal{C}'$  in their interior. This way we obtain the collection  $\mathcal{C}''$  of  $x$ -monotone curves that do not intersect in their interior. The *arrangement*  $\mathcal{A}(\mathcal{C})$  of the curves in  $\mathcal{C}$  is the planar map induced by the curves in  $\mathcal{C}''$ .

<sup>2</sup> <http://www.mpi-sb.mpg.de/LEDA/leda.html>

<sup>3</sup> We define an  $x$ -monotone curve to be a curve that is either a vertical segment or a Jordan arc such that any vertical line intersects it in at most one point.



**Fig. 1.** Arrangements of segments (a), circles (b), polylines (c) and canonical-parabolas (d).

The arrangement package (described in Section 3) is built as a layer above CGAL's *planar map* package [5,10] adding considerable functionality to it. The planar map package supports planar maps of general  $x$ -monotone curves that do not intersect in their interior. It does not assume connectivity of the map (i.e., holes inside faces are supported), and enables different strategies for efficient point location. The representation used in our arrangement package enables easy traversal over the planar map induced by the arrangement, while maintaining the information of the original input curves. For example, traversal over all edges in the planar map that originate from the same curve is easily achieved. Like the planar map package, it supports holes and point location and vertical ray shooting queries. Using it, we have implemented arrangements of different curves: line segments, circle arcs, polylines and others (see Figure 1).

We also present (in Section 4) an application based on this package of an adaptive data structure for efficient point location in arrangements of algebraic parametric curves, provided they meet certain conditions. An example is given for arrangements of quadratic Bézier curves. The underlying idea is to approximate the Bézier curves with enclosing polygons and do the operations on these polygons. If the enclosing polygons do not enable us to solve the problem, a subdivision is performed on the polygons which gives a finer approximation of the original Bézier curves. We resort to operations on the actual Bézier curves only if we cannot resolve it otherwise (or if we pass some user-defined threshold). In this sense our scheme resembles the adaptive arithmetic schemes [11,22] that work with floating point approximations (such as interval arithmetic), and resort to (slow) exact arithmetic only when the computation cannot be resolved otherwise. Indeed our work was inspired by these schemes. The code for the software described in this paper, as well as more elaborate explanations of the algorithms can be found in <http://www.math.tau.ac.il/~hanniel/ARRG00/>.

## 2 Preliminaries

### 2.1 Planar Maps in CGAL

The data structure for representing planar maps in CGAL supports traversal over faces, edges and vertices of the map, traversal over a face and around a

vertex and efficient point location. The design is flexible enabling the user to define his/her own special curves as long as they support a predefined interface. A detailed description of the design and implementation of the planar maps package in CGAL can be found in [10].

In CGAL all algorithms and data structures of the basic library are passed an additional template parameter. This parameter, the so-called *traits* class, makes it possible to use the algorithms in a flexible manner. The traits class is a concept<sup>4</sup> that defines the geometric interface to the class (or function). The set of requirements defining it are given in the documentation of the class. We have formulated the requirements from the planar map's traits, so they make as little assumptions on the curve as possible (for example, linearity of the curve is not assumed). This enables the users to define their own traits classes for different kinds of curves that they need for their applications.

The planar maps package provides a mechanism (the so-called *strategy pattern* [12]) for the users to implement their own point location algorithm. Given a query point  $q$  the algorithm will return the face in the planar map where  $q$  is located. We have implemented three point location strategies ranging from a naive one to a (theoretically) very efficient one. Each strategy is preferable in different situations, the trade-offs between them are discussed in [5,10].

## 2.2 Computing the Combinatorial Structure of Arrangements of Parametric Curves

Neagu and Lacolle [20] describe an algorithm for computing the combinatorial structure of arrangements of curves that satisfy certain conditions. In the following paragraphs we summarize some of their theoretical results which we use in our work. The curves they deal with are a family of piecewise convex curves, where each curve  $C$  fulfills the following requirements: (i) The polygonal line  $P = P_0P_1\dots P_m$ , named the *control polygon* of the curve  $C$ , is simple and convex. (ii)  $P_0$  and  $P_m$  are the extremities of  $C$ . (iii) The line  $P_0P_1$  is tangent to  $C$  in  $P_0$  and the line  $P_{m-1}P_m$  is tangent to  $C$  in  $P_m$ . (iv) The curve is included in the convex hull of its control polygon. (v) The curve and its control polygon satisfy the variation diminishing property (i.e., any line that intersects  $C$  at  $k$  points intersects the control polygon in at least  $k$  points, see [9]). (vi) There exists a subdivision algorithm through which the control polygon converges to the curve. Since the control polygon is convex, the variation diminishing property implies that these curves are convex. These conditions can be satisfied by many well-known parametric curves: Bézier and rational Bézier curves, B-splines and NURBS<sup>5</sup>.

We follow the terminology introduced in [20]. The *carrier polygon* of a curve is the segment between the two endpoints of the control polygon. Therefore,

<sup>4</sup> By *concept* we mean a class obeying a set of predefined requirements, which can be therefore used in generic algorithms; see [3] for more information on concepts in generic programming.

<sup>5</sup> As we show in the full version of this paper, theoretically, these conditions are satisfied by many types of convex “well-behaved” curves.

these two polygonal lines, the control polygon and the carrier polygon bound the curve between them. After a subdivision has taken place, the union of the subcurves is the original curve. In some cases when we refer to the carrier (resp. control) polygon of a curve we mean the union of the carrier (resp. control) polygons of its subcurves. It is easy to see that these also bound the original curve.

In order to isolate the intersection points of two curves, we use the results obtained by Neagu and Lacolle [20] – they give sufficient conditions on the control and carrier polygons of two subcurves which guarantee that the subcurves intersect exactly once. In our application we verify these conditions in order to isolate vertices of the arrangement (see Section 4).

### 3 Arrangements in CGAL

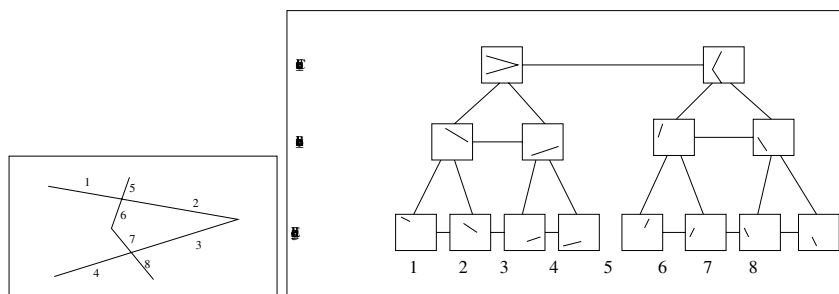
The arrangement package is a layer built on top of the planar map layer. Given a set  $\mathcal{C}$  of (not necessarily  $x$ -monotone and possibly intersecting) curves, we wish to construct the planar map induced by  $\mathcal{C}$ . However, we would like to do so without loss of information. For example, we want to be able to trace all edges in the planar map originating from the same curve of  $\mathcal{C}$ . Furthermore, we want the users to be able to control the process of decomposing the curves into  $x$ -monotone curves to be inserted into the planar map. This is achieved with a special data structure we call a *curve hierarchy tree*.

The package does *not* assume general position. In particular it supports  $x$ -degenerate input (e.g., vertical line segments), non  $x$ -monotone curves and overlapping curves. Non  $x$ -monotone curves are partitioned into  $x$ -monotone subcurves and then inserted into the planar map. The fact that two curves overlap is traced by the traits intersection function, before the insertion into the planar map. Thus, given an edge  $e$  in the planar map, the user can traverse all the overlapping subcurves that correspond to  $e$ . The arrangement package is available as part of release 2.1 of the CGAL library.

**Hierarchy Tree.** When constructing an arrangement for a set  $\mathcal{C}$  of curves we decompose each curve  $C \in \mathcal{C}$  in two steps obtaining the collections  $\mathcal{C}'$  and  $\mathcal{C}''$ . We can regard these collections as levels in a hierarchy of curves where the union of the subcurves in each level is the original curve  $C$ . In some applications we might wish to decompose each curve into more than three levels. For example, in an arrangement of generalized polygons, the boundary of each is a piecewise algebraic curve, we may want to decompose the curve into its algebraic subcurves and only then make them  $x$ -monotone. We enable the users to control this process by passing their own *split functions* as parameters. We make use of this feature in the application described in Section 4.

We store these collections in a special structure, the *hierarchy tree*. This structure usually consists of three levels, although in some cases it consists of less (e.g., when inserting an  $x$ -monotone curve) or more. The standard levels are: (i) Curve node level: the root of the tree – holds the original curve. (ii)





**Fig. 2.** A simple arrangement of two polylines, and its corresponding hierarchy tree (the edges are numbered according to their order in the tree).

Subcurve node level: inner nodes of the tree – hold the subcurves of the original curve after decomposition. In default mode these are  $x$ -monotone curves. (iii) Edge node level: leaves of the tree – hold the curves corresponding to the edges of the planar map induced by the arrangement. Figure 2 shows an example of a simple arrangement of polylines and its corresponding curve hierarchy.

**Geometric Traits.** There are several functions and predicates required by the arrangement class that are not required in planar maps. In our implementation this means that the requirements from the arrangement traits class are a superset of those of the planar map traits. The main functions that are added are for handling intersections of  $x$ -monotone curves and for detecting and handling non  $x$ -monotone curves and splitting them into  $x$ -monotone subcurves. For the latter a new, more general, `Curve` type is required since in the planar map only  $x$ -monotone curves were handled. The full set of requirements is given in the arrangement documentation [5].

We have implemented several traits classes for different curves. Traits classes for line segments and circle arcs can be found in the CGAL-2.1 distribution. We have also implemented traits classes for polygonal lines and for canonical parabola arcs. Figure 1 shows examples of arrangements constructed with the segment, circle, polyline and canonical-parabola traits. The programs for generating them can be found in the website mentioned above [16].

## 4 Adaptive Point Location

The package described in Section 3 can perform point location on arrangements of any curves, provided they satisfy the requirements of the traits class. However, in practice, for algebraic curves and especially for parametric algebraic curves, implementing these predicates and functions robustly is difficult, requiring symbolic algebraic techniques. Even when this is feasible the time penalty for these methods is very high. We now describe an application of the arrangement package that aims to avoid these computations as much as possible, relying on rational arithmetic only.

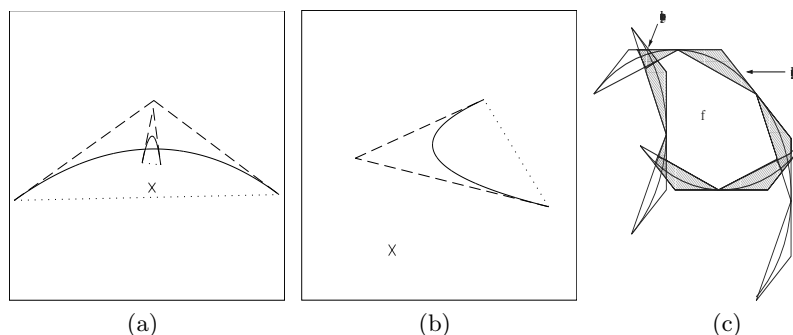
The application described in this section addresses the following problem: Given a set  $\mathcal{C}$  of curves satisfying the conditions presented in Section 2.2, we wish to build an efficient adaptive data structure for point location and vertical ray shooting queries. The algorithm we describe is static (i.e., all the curves are given in the initialization step).

The general scheme of our algorithms is to bound the curves by a *bounding polygon*, and perform all operations on this polygon rather than on the curve itself. If the bounding polygon's approximation of the curve is not sufficient, we refine it by a subdivision process. This scheme is similar to the one presented by Neagu and Lacolle. Like them, we have also implemented our algorithm on Bézier curves (in our case on quadratic Bézier curves) which satisfy the given conditions and have an easy-to-implement subdivision scheme. However, there are several significant differences between our work and the work presented in [20]. Unlike them, we construct an efficient data structure for point location in the arrangement. Furthermore, our algorithm is local and adaptive whereas theirs is global. That is, we only aim to refine the faces of the arrangements that are needed by a query. Another difference is that we do not make an assumption that the arrangements are connected like they do, i.e., we support holes inside the faces of the arrangement; this in turn raises considerable algorithmic difficulties. We also implement a heuristic that traces degenerate cases and deals with them, therefore the input is not assumed to be non-degenerate. There is also a difference from a system point of view. Our application constructs a framework for future work and can easily be extended to other curves (and other representations) by changing the traits class. It is based on CGAL's arrangement package and can therefore benefit from improvements in that package.

#### 4.1 The Algorithms

We have implemented several algorithms for point location and vertical ray shooting queries in the arrangement package. This was done via the point location strategy mechanism. These algorithms work on any curve that conforms to the traits class requirements. In particular, they work on arrangements of line segments and polygonal lines. We can thus use these procedures – we construct the arrangement of bounding polygons of the curves in  $\mathcal{C}$  and perform queries on it; we use the results of these queries to answer the queries on the original curves. The following sections describe the algorithms we use for vertical ray shooting and for point location. In a standard planar map, represented say by a *Doubly Connected Edge List* (DCEL) structure [6, Chapter 2], an answer to a vertical ray shooting query is easily transformed into an answer to a point location query. In our adaptive setting however, point location queries are much more involved than vertical ray shooting queries as we explain below.

**Vertical Ray Shooting.** Given a query point  $q$  in the plane, we wish to find the curve  $C_i \in \mathcal{C}$  that is directly above  $q$  (i.e., is above  $q$  and has the minimum vertical distance to it). We need to find sufficient conditions on the bounding polygons which guarantee that  $C_i$  is the result of the query from  $q$ .



**Fig. 3.** (a) A global scheme for intersection detection is needed: the intersection of the large triangle with the small one will not be detected by a local intersection detection scheme, and the vertical ray shooting query will return the small curve instead of the large one. (b) The second condition is necessary although the first condition is met: the carrier polygon is not above the query point and neither is the original curve. (c) A face with boundary and intersection polygons.

The first condition is that the bounding polygon directly above  $q$  (i.e., the result of the vertical ray shooting query in the arrangement of bounding polygons) does not intersect any other polygon. A local scheme that goes over the border of the bounding polygon and checks for intersections with other segments is insufficient (see Figure 3(a)). We need a global scheme to detect intersections between bounding polygons. This is achieved using an *intersection graph*  $IG = (V, E)$ : every node  $v \in V$  of the graph represents a bounding polygon of a subcurve in the arrangement and two nodes are connected by an edge  $e \in E$  if the bounding polygons they represent intersect. The intersection graph is initialized at the construction of the arrangement. When a subdivision is performed not all the nodes of the graph are effected. Only the nodes that are adjacent to the node that corresponds to the subdivided polygon need to be updated. The second condition is that both the control and carrier polygons of the bounding polygon are above the query point. If this condition is not met then we cannot guarantee that the original curve is above the query point (see Figure 3(b)).

Our algorithm performs a vertical ray shooting query in the bounding polygon arrangement. It then checks for the sufficient conditions on the result bounding polygon  $bp_r$  and if they are met then the answer to the query is found and we report it. Otherwise, it performs a subdivision on  $bp_r$  and on the bounding polygons that intersect it. In every subdivision of a polygon the intersection graph is updated.

**Point Location.** Given a query point  $q$  in the plane, our algorithm will find the face  $f$  of the curve arrangement where  $q$  is located. The output of the algorithm will be the ordered lists of curves (a list for the outer boundary and lists for inner boundaries if they exist) that contribute to the boundary of the face.

For lack of space we do not give the details of the algorithm here. We refer the reader to the full version of the paper (see [16] in the related website). The description consists of how to find the boundary of a simply connected face. Then the algorithm is extended to faces with holes, and finally we describe how the algorithm handles degeneracies.

We have implemented the algorithms for adaptive point location, using the arrangement class described in Section 3, which is named `Arrangement_2`. The class `Adaptive_arr` derives from the class `Arrangement_2`. Its traits class is a superset of the polyline traits for `Arrangement_2`. In its constructor the class gets a sequence of curves and inserts their bounding polygons into the arrangement. It then initializes the intersection graph (in our current implementation we do this with a naive algorithm). Our application uses the “walk” point location strategy [10]. Currently this strategy gives the best experimental results.

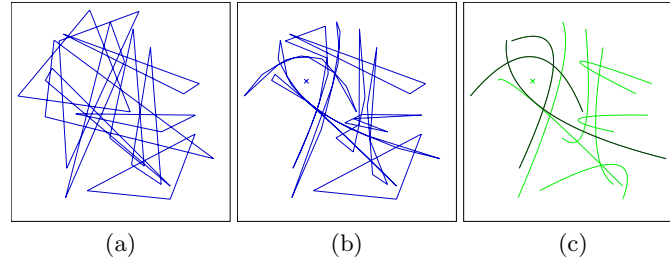
## 5 Experimental Results

We have conducted several experiments on the adaptive point location application<sup>6</sup>. The application is built over the arrangement package using polyline traits. Our main goals in implementing the package were robustness, genericity and flexibility. The code has not been optimized yet. Work is currently underway to speed-up the performance of the package.

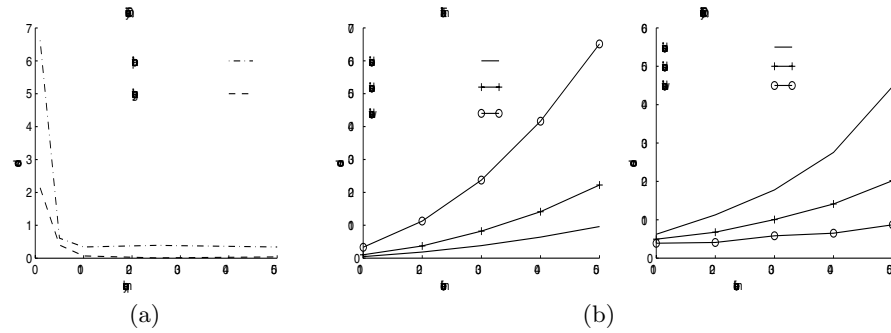
There are a lot of variables which influence the performance of our adaptive point location application. The number of curves and the size of the output (the returned face) are the obvious ones. Apart from them the performance of our adaptive scheme is influenced by the density of the arrangement – if the curves are dense then more subdivisions need to be performed in order to isolate their intersection points. Another parameter is the distance of the query point from the curve, the closer the point is to a curve the larger the number of subdivisions we may need to perform. An important parameter is the number of queries we perform and their relative location. Since our algorithm is adaptive, a query in a face that has already been queried should be faster than the first query in that face. Similarly, a query close to a face that has already been queried is also anticipated to be faster since some of the subdivisions have already been performed for the neighboring face.

We have constructed test inputs of ten random sample sets of quadratic Bézier curves (using CGAL’s random triangle generator). Figure 4 shows such a set with 10 curves and the corresponding segment arrangement before and after the first point location query. The curves of the located face are marked in bold line in the Bézier arrangement. We have run the input curves on 50 random points distributed evenly in a disk centered at the origin with a radius of 100 units (the curves are distributed in a circle of radius 400). Figure 5(a) shows the average time per query as a function of the number of queries already performed,

<sup>6</sup> All experiments were done on a Pentium-II 450Mhz PC with 528MB RAM memory, under Linux.



**Fig. 4.** An arrangement of 10 Bézier curves and its corresponding segment arrangement before (a) and after (b) the first point location; (c) displays in bold line the curves that bound the face containing the query point.



**Fig. 5.** (a) The decrease in the average query time as a function of the number of queries. (b) The trade-off between subdivision at the initialization step and at the queries.

for ten curves. We can see clearly that the time reduces considerably as we make more queries.

There is a trade-off between the initialization step and the rest of the algorithm. If we subdivide the initial control polygon in the initialization step we prolong this step; however, since the resulting bounding polygon is closer to the curve the query will take less time. We timed the initialization step and Figure 5(b) demonstrates the first point location query (as Figure 5(a) demonstrates, this is the significant query) as a function of the number of curves in the arrangement. We repeated the procedure, performing one and two subdivisions in the initialization step. As can be expected the time for the point location query reduces while the initialization time increases as more subdivision steps are performed at the preprocess step. Implementing this change amounted to changing a few lines in a single function of the traits class, therefore the users can experiment with this trade-off for their needs.

## 6 Conclusions and Future Work

We presented a robust, generic and flexible software package for 2D arrangements of general curves and an application based on it for adaptive point location in arrangements of parametric algebraic curves. We have implemented this application for quadratic Bézier curves, and presented some experimental results.

The work described in this paper is a framework that can be extended and further improved. Work on improving and speeding-up the arrangement package is currently underway in two main directions: improving the internal algorithms (e.g., the algorithms for inserting new curves into the arrangement) and implementing new traits classes (e.g., traits classes that make use of filtering schemes, and for additional types of curves). The adaptive point location application will also benefit from these improvements. The adaptive point location can also be applied to other parametric curves such as splines. Implementing traits classes for these curves is also left for future work. Two software packages are currently being developed on top of our arrangement package: one for map overlay and boolean operations and the other for *snap rounding* [14] arrangements of segments.

## Acknowledgment

The authors wish to thank Sigal Raab, Sarel Har-Peled, Oren Nechushtan, Eyal Flato, Lutz Kettner and Eti Ezra for helpful discussions concerning the package described in this paper and other contributions. The authors also thank Manuella Neagu and Bernard Lacolle for providing us with their full paper.

## References

1. N. Amenta. Directory of computational geometry software. <http://www.geom.umn.edu/software/cglist/>.
2. N. Amenta. Computational geometry software. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 52, pages 951–960. CRC Press LLC, Boca Raton, FL, 1997.
3. M. Austern. *Generic Programming and the STL – Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.
4. H. Brönniman, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. Technical Report MPI-I-98-1-030, Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany, 1998.
5. *The CGAL User Manual, Version 2.1*, Jan. 2000. <http://www.cs.uu.nl/CGAL>.
6. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Heidelberg, Germany, 1997.
7. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

8. A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.
9. G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, 3rd edition, 1993.
10. E. Flato, D. Halperin, I. Hanniel, and O. Nechushtan. The design and implementation of planar maps in CGAL. In *Proc. of the 3rd Workshop of Algorithm Engineering*, volume 1668 of *Lecture Notes Comput. Sci.*, pages 154 – 168. Springer-Verlag, 1999.
11. S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
13. M. Goldwasser. An implementation for maintaining arrangements of polygons. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C32–C33, 1995.
14. M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.
15. D. Halperin. Arrangements. In J. E. Goodman and J. O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 21, pages 389–412. CRC Press LLC, Boca Raton, FL, 1997.
16. I. Hanniel. The design and implementation of planar arrangements of curves in CGAL. M.Sc. thesis, Dept. Comput. Sci., Tel Aviv University, Tel Aviv, Israel. Forthcoming, <http://www.math.tau.ac.il/~hanniel/ARRG00/>.
17. J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: a library for efficient manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 360 – 369, 1999. <http://www.cs.unc.edu/~geom/MAPC/>.
18. K. Mehlhorn, S. Näher, C. Uhrig, and M. Seel. *The LEDA User Manual, Version 4.1*. Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany, 2000.
19. K. Melhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
20. M. Neagu and B. Lacolle. Computing the combinatorial structure of arrangements of curves using polygonal approximations. Manuscript. A preliminary version appeared in Proc. 14th European Workshop on Computational Geometry, 1998.
21. M. Sharir and P. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, 1995.
22. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.
23. S. Skiena. *The Algorithm Design Manual*. Telos/Springer-Verlag, 1997. <http://www.cs.sunysb.edu/~algorithm/index.html>.

# Planar Point Location for Large Data Sets: To Seek or Not to Seek

Jan Vahrenhold and Klaus H. Hinrichs

FB 10, Institut für Informatik, Westfälische Wilhelms-Universität, Einsteinstr. 62,  
48149 Münster, Germany. {jan,khh}@math.uni-muenster.de

**Abstract.** We present an algorithm for external memory planar point location that is both effective and easy to implement. The base algorithm is an external memory variant of the bucket method by Edahiro, Kokubo and Asano that is combined with Lee and Yang’s batched internal memory algorithm for planar point location. Although our algorithm is not optimal in terms of its worst-case behavior, we show its efficiency for both batched and single-shot queries by experiments with real-world data. The experiments show that the algorithm benefits from its mainly sequential disk access pattern and significantly outperforms the fastest algorithm for internal memory.

## 1 Introduction

The well-known problem of *planar point location* consists of determining the region of a planar subdivision that contains a given query point. We assume that a planar subdivision is given by  $N$  line segments, and that each segment is labeled with the names of the two regions it separates. In this setting, a point location query can also be answered by *vertical ray shooting*, i.e., by extending a vertical ray from the query point towards  $y = +\infty$  and reporting the first segment hit by that ray. In this paper, we are interested in the problem of locating points in a very large planar subdivision that is stored on disk while reducing the overall time spent on I/O operations. Our model of computation is the standard two-level I/O model proposed by Aggarwal and Vitter [3]. In this model,  $N$  denotes the number of elements in the problem instance,  $M$  is the number of elements fitting in internal memory, and  $B$  is the number of elements per disk block, where  $M < N$  and  $2 \leq B \leq M/2$ . An I/O is the operation of reading (or writing) a disk block from (into) external memory. Computations can only be done on elements present in internal memory. Our measures of performance are the number of I/Os used to solve a problem and the amount of space (disk blocks) used. Aggarwal and Vitter [3] considered sorting and related problems in the I/O model and proved that sorting requires  $\Theta\left((N/B) \log_{M/B}(N/B)\right)$  I/Os. When dealing with massive data sets, it is likely that there are  $K$  point location queries to be answered at a time—also called *batched point location*.

Most algorithms for planar point location exploit hierarchical decompositions which can be generalized to a so-called *trapezoidal search graph* [1]. Using balanced hierarchical decompositions, searching then can be done efficiently in



both the internal and external memory setting. As the query points and thus the search paths to be followed are not known in advance, external memory searching in such a graph will most likely result in unpredictable access patterns and random I/O operations. Disk technologies and operating systems, however, support sequential I/O operations much more efficiently than random I/O operations.

Experimental results published in the database literature [5,17] reveal that external memory algorithms can benefit even from relatively small clusters of sequentially stored data. We will show how to obtain an algorithm for solving the external memory planar point location problem that fulfills this criterion.

**Previous Results.** The problem of internal memory planar point location has been studied extensively, and we refer the reader to general surveys on that topic [22,24]. We now briefly review related work on external memory planar point location and introduce the notation  $n = N/B$ ,  $m = M/B$ , and  $k = K/B$ .

Goodrich et al. [16] suggested a batched planar point location algorithm for monotone subdivisions using persistent  $B$ -trees that needs  $\mathcal{O}((n + K/B) \log_m n)$  I/O operations for preprocessing  $N$  segments and locating  $K$  query points. Arge, Vengroff, and Vitter [7] perform batched planar point location of  $K$  points among  $N$  segments in  $\mathcal{O}((n + K/B) \log_m n)$  I/O operations using the *extended external segment tree*. The algorithm relies on an external memory variant of fractional cascading [9], and thus the practical realization of their algorithm is rather complicated. Conceptually, the algorithm allows for single-shot queries.

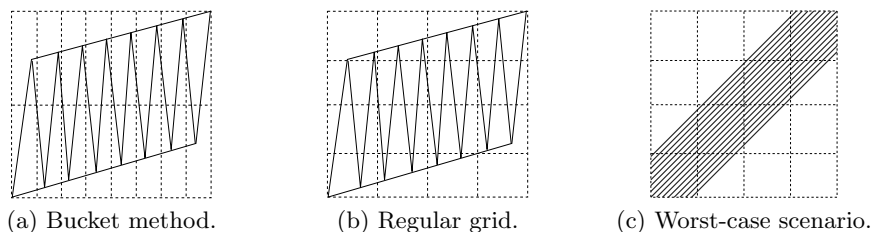
Crauser et al. [11] merge the batch filtering technique of Goodrich et al. [16] into an external variant of Mulmuley's randomized incremental construction of the trapezoidal decomposition [20] and obtain an  $\mathcal{O}((n + K/B) \log_m n)$  batched randomized algorithm. Arge and Vahrenhold [6] presented an algorithm for single-shot point location in a dynamically changing general subdivision building upon an earlier result by Agarwal et al. [2]. The space requirement is linear, queries and insertions take  $\mathcal{O}(\log_B^2 n)$  I/O operations, and deletions can be performed in  $\mathcal{O}(\log_B n)$  I/O operations. The bound for queries is worst-case whereas the update complexity is amortized.

One of the most interesting open problems related to point location is to generalize any of the above algorithms to three or higher dimensions.

## 2 A Practical External Memory Algorithm

The I/O-efficient algorithms mentioned in the previous section employ rather complicated constructs that mimic internal memory data structures and are hard to realize. For practical applications, however, it is often desirable to trade asymptotically optimal performance for simpler structures if there is hope for comparable or even faster performance in practice.

Our algorithm for performing batched planar point location in an  $N$ -segment planar subdivision is a modification of the bucket method proposed by Edahiro, Kokubo, and Asano [13]. In this section, we describe the basic algorithm and show how to adapt the algorithm to the external memory setting such that both single-shot and batched point location queries can be processed effectively.



**Fig. 1.** Partitions induced by grid methods ([13], modified).

## 2.1 Description of the Basic Algorithm

The bucket method partitions the minimum bounding box of the subdivision into  $\mathcal{O}(N)$  equal-sized buckets and assigns to each bucket only the part of the subdivision falling into it. If an original segment crosses one or more grid lines, it is split into a corresponding number of fragments. These splits lead to a replication of the data, and in the worst case we might end up with a super-linear number of fragments. A similar method has been developed independently by Franklin [15] who used a regular grid to compute intersections of line segments.

To answer a point location query, the query point is hashed into the bucket it is contained in by normalizing its coordinates using the *floor*-function. Then comparisons have to be performed only against the portion of the map contained in this bucket. Let  $x_d$  be the width and  $y_d$  be the height of the minimum bounding box of the map, and let  $(x_i^{(1)}, y_i^{(1)})$  and  $(x_i^{(2)}, y_i^{(2)})$  denote the end-points of segment  $i$ . Then let  $S_x := \left( \sum_{i=1}^N |x_i^{(2)} - x_i^{(1)}| \right) / x_d$  and let  $S_y := \left( \sum_{i=1}^N |y_i^{(2)} - y_i^{(1)}| \right) / y_d$ . Two constants  $\alpha_x$  and  $\alpha_y$  are chosen such that they satisfy  $\alpha_x / \alpha_y = S_y / S_x$  and  $\alpha_x \alpha_y = 1$ . Then the numbers of rows and columns are chosen as  $N_x = \lfloor \alpha_x \sqrt{N} \rfloor$  and  $N_y = \lfloor \alpha_y \sqrt{N} \rfloor$  for a total of  $\mathcal{O}(N)$  buckets. An example for partitioning a map is shown in Figure 1(a)—compare this to Franklin’s regular  $\sqrt{N} \times \sqrt{N}$  grid shown in Figure 1(b).

Each bucket of the partition stores information about the segments that fall within the bucket or intersect its border. The analysis of the algorithm shows a space requirement of  $\mathcal{O}(L)$  where  $L$  is the total number of fragments into which the original segments are broken by the partitioning lines, i.e., the total number of fragments in the buckets. In the worst case,  $L$  can be on the order of  $N\sqrt{N}$  [26] (see also Figure 1(c)) which leads up to  $\mathcal{O}(N)$  query time, but for real-world data sets, we have  $L \in \mathcal{O}(N)$  and expect  $\mathcal{O}(1)$  query time as is confirmed by practical results [8,13].

This algorithm heavily relies on the *floor*-function which is not an integral component of the Real RAM model of computation. In our situation, however, we can simulate this function on the Real RAM by a binary search on the rows and columns, thereby increasing the query time to  $\mathcal{O}(\log_2 \sqrt{N}) = \mathcal{O}(\log_2 N)$ . This query time matches the worst-case complexity of several optimal algorithms for internal memory planar point location [14,18,23].

## 2.2 The External Version of the Basic Algorithm

The key to fast and simple external memory planar point location is the observation that each two dimensional grid can be linearized. As we shall demonstrate, this is why both preprocessing and point location can be reduced to a single sorting step and a very small number of linear scans. Since there exists a variety of fast sorting algorithms and since disks are optimized for sequential I/O operations, this is a very desirable property for an external memory algorithm.

To provide the  $\mathcal{O}(n\sqrt{n})$  external memory analogon to the  $\mathcal{O}(N\sqrt{N})$  worst-case space bound, we need to move from a grid with  $\mathcal{O}(N)$  buckets to an  $n_x \times n_y$  grid with  $\mathcal{O}(n)$  buckets where  $n_x := \lfloor \alpha_x \sqrt{n} \rfloor$  and  $n_y := \lfloor \alpha_y \sqrt{n} \rfloor$ .

**Preprocessing the Data.** Given an  $N$ -segment planar subdivision, we scan all  $N$  segments in  $\mathcal{O}(n)$  I/O operations and compute the granularity of the grid internally as in the basic algorithm. The main problem when transforming the algorithm into an external memory variant is how to store the subdivision in the buckets. To reduce the number of disk accesses for storing the subdivision, it is our primary goal to group the segments in such a way that each bucket is touched at most  $\mathcal{O}(1)$  times. Following the approach of Franklin [15], we traverse the segments a second time and label each segment with the number of the bucket it falls into. If a segment crosses a grid line, it is split into fragments. Each fragment is labeled with the number of its containing bucket. After we have processed all segments, we use an optimal external sorting algorithm to sort the segments and fragments according to their labels. Then we scan the sorted list and transfer the segments and fragments blockwise to the buckets they belong to. Since each grid line can split a single segment into at most two fragments, and we have a total of  $\mathcal{O}(\sqrt{n})$  grid lines and  $N$  segments, we get the following result on the preprocessing:

**Lemma 1.** *If an  $N$ -segment polygonal map is broken into  $\mathcal{O}(N)$  fragments, a data structure for external memory planar point location can be constructed in  $\mathcal{O}(n \log_m n)$  I/O operations. In the worst case, i.e., if the map is broken into  $\mathcal{O}(N\sqrt{n})$  fragments, the number of I/O operations is  $\mathcal{O}(n\sqrt{n} \log_m n)$ .*

The only external memory operations used during preprocessing are linear scans and external sorting, and thus we will not encounter random I/O operations except for the merging phase of sorting. Except for very unlikely configurations, the buckets will not contain exactly the same number of fragments each, and several buckets might even contain no fragments at all. Hence, the straightforward approach of assigning the same amount of disk space to each bucket, i.e., the required number of blocks to hold the largest bucket, wastes space. Instead, we store all fragments grouped by the bucket they belong to sequentially in one single data stream. Access to the buckets is granted by indices stored in a separate index stream that can be regarded as an external memory array for pointers into the data stream. The buckets are arranged in the data stream according to a linearization of the two-dimensional grid in row-by-row order.

**Batched Point Location.** To perform the batched planar point location of  $K$  points, we label each point with the name of the bucket it falls into and sort the points according to these labels. Then we scan the sorted list, and for

each relevant bucket, we load the corresponding portion of the map into main memory where we invoke an internal memory algorithm. To avoid uncontrolled I/O operations due to paging, the size of each bucket in question clearly should not exceed the available main memory size. As will be discussed in Section 3, it is most unlikely that—even for systems with small parameter values  $M$ —there will be a real-world data set which causes a single bucket to overflow. However, if there is an overfull bucket, we can detect it during preprocessing, and can further preprocess the bucket (or rather, the column containing it) with an optimal external memory point location algorithm, e.g., the algorithm of Arge et al. [7]. We omit details and only note that this additional preprocessing does not asymptotically increase the overall preprocessing time [26].

The costs for locating  $K$  points are composed of the I/O operations spent on sorting and on loading the buckets. The number of buckets to be loaded depends on the distribution of the actual query points. Therefore, we assume for our analysis a “malicious” setting in which the query points fall into  $\Theta(\min\{n, K\})$  different buckets. As discussed above, the number of blocks occupied by a single bucket is  $\mathcal{O}(1)$  in practice, and then we can load all relevant buckets in  $\Theta(\min\{n, K\})$  I/O operations.

If a bucket with  $N'$  fragments fits into main memory, the I/O costs for locating  $K'$  query points in this bucket are linear in  $K'/B$  and  $N'/B$ , since we can process the query points blockwise as described in Section 2.2. However, if the bucket does not fit into main memory, we have to locate the  $K'$  query points using the extended external segment-tree which takes  $\mathcal{O}(((K' + N')/B) \log_m (N'/B))$  I/O operations [7]. Since  $\Theta(\sqrt{n})$  buckets might contain  $\Theta(N)$  fragments each in the worst case, the overall I/O complexity for locating  $K$  query points within their containing buckets can be as bad as  $\Theta(k \log_m n + \min\{\sqrt{n}, K\} \cdot n \log_m n)$ .

**Lemma 2.** *The batched planar point location for  $K$  points in an  $N$ -segment planar subdivision can be effected in  $\mathcal{O}(k \log_m k + \min\{n, K\})$  I/O operations if each bucket contains  $\mathcal{O}(B)$  fragments. In the worst case, i.e., if the segments are broken into  $\mathcal{O}(N\sqrt{n})$  fragments, the number of I/O operations is in  $\mathcal{O}(k \log_m k + k \log_m n + \min\{\sqrt{n}, K\} \cdot n \log_m n)$ .*

If there are no buckets that store an extended external segment-tree, the only external memory operations we use are linear scans and external sorting. Details about how to handle empty buckets and how to answer vertical ray shooting queries (as opposed to point location queries) are given in [26].

**Single-Shot Point Location.** Locating a single point can obviously be performed by transforming the coordinates of the query point into the coordinates of the corresponding data bucket by normalization. In practice, a single bucket contains only  $\mathcal{O}(B)$  segments, so the portion of the map stored in this bucket is loaded into main memory in order to perform internal memory point location. In a worst-case scenario, the  $\Omega(M)$  fragments in the bucket are organized by an extended external memory segment tree as described above. Locating a single point in that tree corresponds to a traversal of a root-to-leaf path. Attached to the nodes along this path are at most  $\mathcal{O}(N)$  fragments, so the worst-case query time for locating a single point is bounded by  $\mathcal{O}(n)$ .

**Lemma 3.** *A single shot query can be performed in  $\mathcal{O}(1)$  I/O operations if each bucket contains  $\mathcal{O}(B)$  fragments. In the worst case, i.e., if a single bucket contains  $\mathcal{O}(N)$  fragments,  $\mathcal{O}(n)$  I/O operations may be necessary to answer a single shot query.*

**Internal Memory Point-Location.** For the internal memory part we use Lee and Yang’s algorithm [19]. This algorithm—a batched variant of Dobkin and Lipton’s *slab method* [12]—performs a plane-sweep with a vertical line  $V$  over the query points and the segments. While sweeping, we maintain all segments intersecting  $V$  sorted by the  $y$ -coordinate of the intersection point. Whenever the line stops at a query point  $p$ , a binary search among the segments intersecting  $V$  is performed to answer the point location query for  $p$ .

We can assume that all  $K_i$  query points for bucket  $i$  are sorted by increasing  $x$ -coordinates since we otherwise can sort them during the initial sorting step at no additional costs. Similarly, we assume that all  $N_i$  segments of bucket  $i$  are presorted by their left endpoint’s  $x$ -coordinate. The space requirement is  $\mathcal{O}(N_i)$  because we do not need to keep all query points in main memory but can read them blockwise from their stream. By using an I/O-optimal algorithm for the worst-case setting, we can guarantee to use no more than  $\mathcal{O}(M)$  internal memory even if there are  $\Omega(M)$  fragments in a single bucket.

**Lemma 4.** *The internal algorithm for locating  $K_i$  points within a bucket storing  $N_i$  segments requires  $\mathcal{O}(N_i)$  internal space and  $\mathcal{O}((N_i + K_i) \log_2 N_i)$  time.*

### 3 Experimental Results

To verify the results of the previous sections, we implemented our algorithm and performed an extensive experimental evaluation with real-world data sets. Due to the sophisticated structure of the other algorithms discussed in Section 1, there exists no implementation of these methods, and thus we are not able to compare our results with an I/O-efficient algorithm. We also do not compare against standard index structures used in spatial databases as these indexes are tree-like structures, and we expect random access patterns similar to those discussed in the introduction. Instead, we compare with the original bucket method of Edahiro et al. that has been identified as the most efficient internal memory point location algorithm for practical data sets with respect to both speed and space requirements [8,13].

**Our Implementation.** We implemented the algorithms in C++ using TPIE [4,27], a collection of templated functions and classes that allow for simple and efficient implementation of external memory algorithms. Since our algorithm basically consists of a sequence of sequential read and write scans, we chose the `stdio`-implementation of TPIE which uses standard UNIX<sup>TM</sup> streams and thus benefits from the operating system’s caching and prefetching mechanisms. We compiled all programs using the GNU C++ compiler (version 2.8.1), with `-O3` level of optimization. All internal memory data structures were realized using the *Standard Template Library* (STL) [21].

All experiments were performed on a Sun<sup>TM</sup> SparcSTATION<sup>TM</sup> 20 Model 51 running Solaris<sup>TM</sup> 2.6 with 32 MB RAM, a 50 MHz SuperSPARC<sup>TM</sup> STP1020A processor, and a Seagate<sup>TM</sup> ST 34520WC (4.2 GB capacity, 9.5 ms average read time, 40 MB/sec. peak throughput) local disk. Motivated by the empirical observation by Chiang [10] that I/O algorithms perform much better when the available main memory is not exhausted completely but only some fraction is used, the memory-management system of TPIE was configured to use no more than 12 MB of main memory at a time. The remaining free memory was available to the block caching mechanism of the operating system.

Our implementation of the internal memory variant was allowed to use all of the available virtual memory, i.e., as much as 480 MB of swap space. To reduce the space requirements of the internal memory variants as much as possible, we modified the original algorithm in two ways. First, we did not require the input data to be present in main memory at the beginning of the algorithm. Instead, we used the stream management of TPIE to scan the input stream in order to compute the grid's granularity. Then, the segments were scanned, broken into fragments, and directly moved to the buckets' lists. The second modification aimed at the data stored within each bucket. In the original variant, each bucket contains a list of segments and two additional lists of pointers to items in this list to guarantee linear running time of the point location algorithm within each bucket. Since the average number of segments per bucket was at most seven and the maximum size of a bucket did not exceed 85 segments (see Table 1), we decided not to maintain these additional lists which would have resulted in allocating additional (virtual) memory but answered the point location query by an  $\mathcal{O}(N \log_2 N)$  plane-sweep algorithm similar to the one described in Section 2.2.

**Characteristics of the Test Data.** We used Digital Line Graph data sets in 1:100,000 scale provided by the U.S. Geological Survey [25]. The characteristics of the data are described in Table 1. Each line segment needs four double-precision floating-point numbers for the coordinates of its endpoints, two integers for identifying the regions on both sides and one integer for the number of the bucket it is assigned to. With a page size of 4,096 bytes, the number  $B_S$  of segments that fit into a disk block is 73. Since each point needs two coordinates and one label, the number  $B_P$  of points that fit into one block is 170.

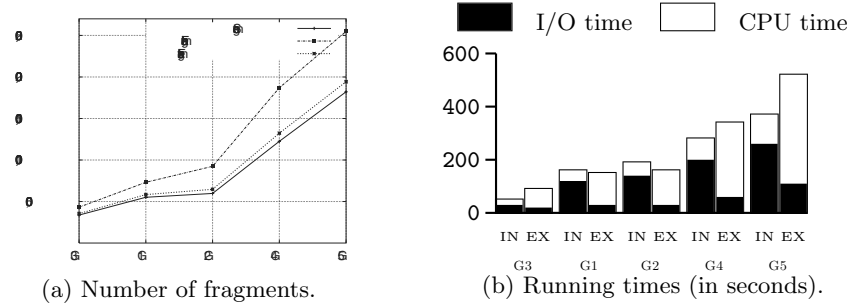
**Grid Construction.** The first set of experiments was dedicated to the preprocessing phase of the point location algorithms. We performed several runs of the preprocessing phase and present the statistics and averaged timings in Figure 2. As can be seen from Figure 2(a), the number of fragments produced by the internal memory algorithm is at most 50% more than the number of original segments. This and the internal memory data structures for the grid and the buckets increase the main memory space requirement by a factor of up to three—note that the algorithm constructs as many buckets as original segments. The penalties for using virtual memory are immense: the algorithm spends more than 2/3 of the overall running time waiting for I/O operations caused by page faults (labeled “I/O time” in Figure 2(b)). In contrast, the space requirement of the external memory variant is very moderate: the number of fragments does not exceed the number of segments by more than 10%. This is true for data sets which produce almost no empty buckets (G1 and G2) and

**Table 1.** Data sets for building the grid (sizes given for raw data).

Data Set	Class	Size	Objects	Buckets		Max. Objects	
				internal	external	int.	ext.
G1 <sup>a</sup>	Hydro	25 MB	550,891	552,123	6,571	37	549
G2 <sup>a</sup>	Roads	27 MB	597,099	598,437	7,189	45	1,199
G3 <sup>b</sup>	Hydro	15 MB	336,555	337,820	4,030	74	747
G4 <sup>b</sup>	Roads	56 MB	1,224,606	1,226,732	14,521	47	908
G5 <sup>a,b</sup>	Roads	83 MB	1,821,705	1,824,589	21,481	85	1,881

<sup>a</sup> Data quadrangles for Albany, Amsterdam, Auburn, Binghampton, Elmira, Glenn Falls, Gloversville, Norwich, Pepacton, Pittsfield, Syracuse, and Utica.

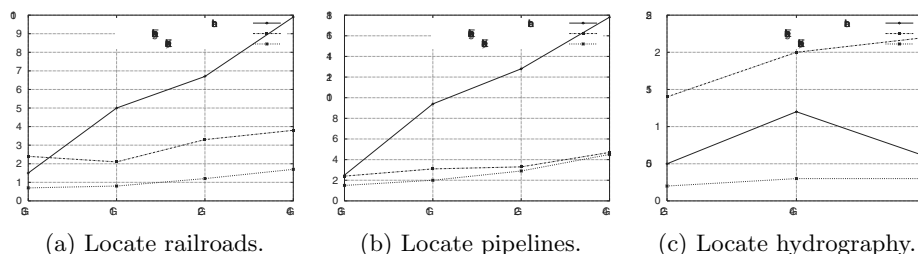
<sup>b</sup> Data quadrangles for Block Island, Bridgeport, Long Branch, Long Island, Middletown, New Haven, Newark, and Trenton.

**Fig. 2.** Constructing the grid in internal (IN) and external (EX) memory.

for data sets which produce a grid with up to 50% empty buckets (G3, G4, and G5). Given  $B_S = 73$ , the average size of non-empty buckets is more than three blocks. With 12 MB internal memory available and 56 bytes required to store a segment,  $M = 12 \cdot 1,024^2 / 56 \approx 224,500$  segments can be packed into one single bucket—compare this to a maximum load of around 1,900 fragments per bucket for our real-world data sets.

The external memory algorithm is accessing the disk for reading, writing, and sorting the data, and the “I/O time” given in Figure 2(b) reflects the time spent waiting for these operations to complete. In contrast to the internal memory algorithm, the external memory variant is sorting the fragments, and the internal memory component of this process contributes a significant amount to the overall running time. What is of much more practical relevance than preprocessing time, is the performance of answering point location queries which we discuss next.

**Single-Shot and Batched Queries.** We performed two sets of point location experiments for each grid where we located the same query points—the vertices of the railroads and pipelines line data for the corresponding area—in single-shot and batched mode. To speed up the single-shot queries, we first use a linear number of tests to determine all segments of the current bucket whose projections onto the  $x$ -axis overlap the  $x$ -coordinate of the query point. If this



**Fig. 3.** Average query time per point (time in milliseconds).

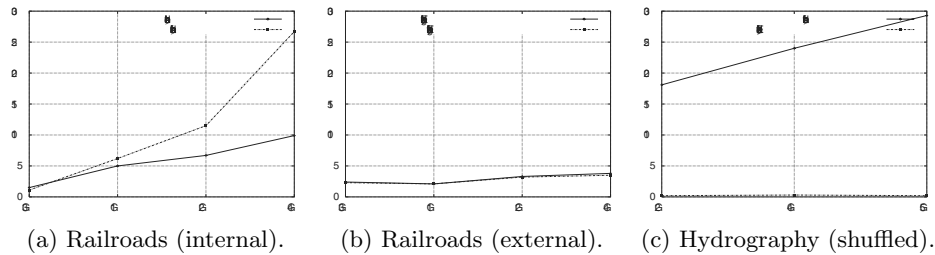
set is empty, we invoke the standard plane-sweep algorithm, making sure that we do not advance past the query point. If there are segments overlapping the query point, we locate the query point in a  $y$ -table containing all these segments.

In our first set of experiments, we used the vertices extracted from the railroads and pipelines data sets as query points. The results presented in Figure 3 show that our proposed batched algorithm outperforms the internal memory algorithm by a factor of up to seven. But even if we compare the running times of single shot queries, we see that the external memory variant is faster by at least a factor of two. The only exception is the data set G3, where the internal memory algorithm is up to 30% faster than the external memory single-shot variant. This comes as no surprise since the data set G3 is by far the smallest data set (see Table 1) and thus the least amount of swap space is needed.

The internal memory algorithm spends more than 95% of the running time on waiting for I/O operations due to paging. Since the hardware we are using has a relatively slow CPU and a relatively fast disk, we expect an even worse performance ratio for systems with faster processors and/or slower disks.

To investigate the behavior of the three algorithms for larger sets of query points, we performed another set of experiments, where we located the vertices extracted from the hydrography data sets in the grids constructed for the roads data sets. The results presented in Figure 3(c) show indeed that the performance of the external memory single-shot algorithm degrades the larger the set of query points is. This can be explained by the fact that now there are up to 80 query points per bucket. The single-shot algorithm needs to perform at least 80 linear scans per bucket, and each bucket contains up to 213 fragments on the average. In contrast, the batched algorithm only performs one plane-sweep per bucket after the initial sorting of the points. As we can see from Figure 3(c), the batched external memory variant is up to eight times faster than the single-shot external memory variant. Since the external memory single-shot variant spends more than 95% of its running time on internal memory operations, it additionally suffers from the relatively slow processor. In contrast, the internal memory algorithm benefits from the relatively fast disk and is able to outperform the external memory single-shot algorithm by a factor of up to four. The batched variant also spends up to 90% of its running time on internal memory operations, but is still able to perform more than three times faster than the internal memory algorithm.





**Fig. 4.** Average query time per point (time in milliseconds).

**Effects of Locality in the Query Pattern.** The data sets used for our experiments store the segment data in form of short polygonal chains, i.e., in sequences of adjacent points. Since we constructed each data set by concatenating several quadrangles of line data, the data files (and hence the extracted point data as well) were stored as a sequence of clusters. In the experiments described above, the query points were processed in exactly the same order. To investigate effects of locality in the query pattern, we repeated all experiments with the query points randomly shuffled (using STL's `random_shuffle` function).

Figures 4(a) and 4(b) contrast the query performance of both single-shot algorithms for clustered and shuffled query points. Since the batched external algorithm sorts the query points according to the buckets they fall into and then processes the sorted sequence, the query performance is not affected by the order of the query points. Hence, we do not repeat the results for the batched variant.

Figure 4(a) shows that the performance of the internal memory algorithm degrades if the locality of the input data is destroyed (again, the smallest data set is an exception). For clustered data, it is likely that the page containing the bucket in question is still in main memory because of a previous request, and in the best case, the cache mechanism of the memory management unit helps speeding up accesses. If the query points—and hence the buckets—are randomly shuffled, the page requests produce not only translation-look-aside buffer (TLB) misses, but in the worst case will cause page faults as well. As a consequence the TLB does not speed up but slow down the algorithm, and we face an increased number of (uncontrolled) I/O operations. Figure 4(a) shows that the performance gets slowed down by a factor of almost three. On the other hand, the I/O-wait is still 97% of the running time. Since the computation done by the internal memory single-shot algorithm is independent from the order of queries, this is a clear indication for a large number of TLB misses.

The performance of the external memory single-shot algorithm, on the other hand, does not seem to depend on the access pattern, and for some situations there is even a minor improvement—see Figure 4(b). This is an indication for I/O operations being overlapped with internal memory computation by the system. The average query time per point is less than 4 ms—this corresponds to less than twice the average single-track seek time of the disk drive. For the internal memory variant, we have up to 26 ms average query time which is more than the full seek time of the disk drive.

Finally, we present a comparison of the internal memory algorithm with the batched external memory variant for locating large sets of query points in shuffled order—see Figure 4(c). This setting reflects the scenario for which our algorithm was designed: locating a large, not necessarily clustered, set of points in a large collection of segments. Again, the internal memory algorithm spends more than 95% of the overall running time waiting for I/O operations, and the average query time per point varies between 18 and 27 ms. Our proposed batched algorithm, on the other hand, answers all point location queries very fast: the average query time per point is at most 0.3 ms. In the best situation, our algorithm is able to perform more than 130 times faster than the internal memory single-shot point location. It should be noted that our algorithm does only temporarily occupy a fixed amount of internal memory. This means that—without the need for paging or swapping—the algorithm can be used on a workstation where other software, e.g., subsystems of a Geographic Information System, needs to reside in main memory.

## 4 Conclusions

We have presented an algorithm for external memory point location along with experimental results of a comparison with the fastest and most space-efficient internal memory method. To our knowledge, this is the first experimental evaluation of an algorithm for external memory point location. The comparison with the fastest and most space-efficient internal memory method shows a clear superiority of the batched algorithm for all settings. The main reason for this superiority is the mainly sequential disk access pattern of our batched algorithm that benefits from operating systems and disk controllers optimized for sequential I/O operations. Due to the mainly sequential disk access pattern, our batched algorithm will be a strong competitor of any I/O-optimal algorithm with respect to performance for real-world data sets. It remains open, though, to verify this conjecture by an experimental evaluation. Finally, note that the design of the algorithm allows for an almost straightforward extension to utilize multiple disks and processors—an important aspect when dealing with large data sets.

## References

1. U. Adamy and R. Seidel. On the exact worst case query complexity of planar point location. *Proc. 9th Annual ACM-SIAM Symp. Discrete Algorithms*, 609–618. 1998.
2. P. Agarwal, L. Arge, G. Brodal, and J. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, 11–20. 1999.
3. A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.
4. L. Arge, R. Barve, O. Procopiuc, L. Toma, D. Vengroff, and R. Wickremesinghe. TPIE user manual and reference, edition 0.9.01a. Duke University, North Carolina, <<http://www.cs.duke.edu/TPIE/>>, 1999. (accessed 12 Jul. 1999).
5. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. Vitter. A unified approach for indexed and non-indexed spatial join. *Proc. 7th Intl. Conf. Extending Databases Technology*, LNCS 1777, 413–429. 2000.

6. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Proc. 16th Annual ACM Symp. Computational Geometry*, 191–200. 2000.
7. L. Arge, D. Vengroff, and J. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Proc. 3rd Annual European Symp. Algorithms*, LNCS 979, 295–310. 1995.
8. K. Baumann. Implementation and comparison of five algorithms for point-location in trapezoidal decompositions. Master's thesis, Fachbereich Mathematik und Informatik, Westfälische Wilhelms-Universität Münster, Germany, 1996. (in German).
9. B. Chazelle and L. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.
10. Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 9(4):211–236, March 1998.
11. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *Proc. 14th Annual ACM Symp. Computational Geometry*, 259–268. 1998.
12. D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.
13. M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency: Comparison with existing algorithms. *ACM Trans. Graphics*, 3(2):86–109, 1984.
14. H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.
15. W. R. Franklin. Adaptive grids for geometric operations. *Proc. 6th Intl. Symp. Automated Cartography (Auto-Carto Six)*, vol. 2, 230–239. 1983.
16. M. Goodrich, J.-J. Tsay, D. Vengroff, and J. Vitter. External-memory computational geometry. *Proc. 34th Annual IEEE Symp. Found. Computer Science*, 714–723. 1993.
17. K. Kim and S. Cha. Sibling clustering of tree-based spatial indexes for efficient spatial query processing. *Proc. 1998 ACM CIKM Intl. Conf. Information and Knowledge Management*, 398–405. 1998.
18. D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.
19. D.-T. Lee and C. Yang. Location of multiple points in a planar subdivision. *Inf. Proc. Letters*, 9(4):190–193, 1979.
20. K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.
21. D. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
22. F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, 2nd edition, 1988.
23. N. Sarnak and R. Tarjan. Planar point location using persistent search trees. *Comm. ACM*, 29:669–679, 1986.
24. J. Snoeyink. Point location. *Handbook of Discrete and Computational Geometry*, Discrete Mathematics and its Applications, chapter 30, 559–574. CRC Press, 1997.
25. U.S. Geological Survey. 1:100,000-scale digital line graphs (DLG). <http://edcwww.cr.usgs.gov/doc/edchome/ndcddb/ndcddb.html> (accessed 26 May 1999).
26. J. Vahrenhold. *External Memory Algorithms for Geographic Information Systems*. PhD thesis, Fachbereich Mathematik und Informatik, Westfälische Wilhelms-Universität Münster, Germany, 1999.
27. D. Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symp.*, 1994.

# Implementation of Approximation Algorithms for Weighted and Unweighted Edge-Disjoint Paths in Bidirected Trees

Thomas Erlebach<sup>1</sup> and Klaus Jansen<sup>2</sup>

<sup>1</sup> Computer Engineering and Networks Laboratory, ETH Zürich, CH-8092 Zürich,  
Switzerland

`erlebach@tik.ee.ethz.ch`

<sup>2</sup> Institute of Computer Science and Applied Mathematics, Technical Faculty of  
Christian-Albrechts-University of Kiel, Olshausenstr. 40, D-24098 Kiel, Germany

`kj@informatik.uni-kiel.de`

**Abstract.** Given a set of weighted directed paths in a bidirected tree, the maximum weight edge-disjoint paths problem (MWEDP) is to select a subset of the given paths such that the selected paths are edge-disjoint and the total weight of the selected paths is maximized. MWEDP is  $\mathcal{NP}$ -hard for bidirected trees of arbitrary degree, even if all weights are the same (the unweighted case). Three different approximation algorithms are implemented: a known combinatorial  $(\frac{5}{3} + \varepsilon)$ -approximation algorithm  $A_1$  for the unweighted case, a new combinatorial 2-approximation algorithm  $A_2$  for the weighted case, and a known  $(\frac{5}{3} + \varepsilon)$ -approximation algorithm  $A_3$  for the weighted case that is based on linear programming. For algorithm  $A_1$ , it is shown how efficient data structures can be used to obtain a worst-case running-time of  $O(m + n + 4^{1/\varepsilon} \sqrt{n} \cdot m)$  for instances consisting of  $m$  paths in a tree with  $n$  nodes. Experimental results regarding the running-times and the quality of the solutions obtained by the three approximation algorithms are reported. Where possible, the approximate solutions are compared to the optimal solutions, which were computed by running CPLEX on an integer linear programming formulation of MWEDP.

## 1 Introduction

Edge-disjoint paths problems have many applications, for example in resource allocation in communication networks. We study approximation algorithms for weighted and unweighted edge-disjoint paths problems in bidirected trees. Bidirected trees are the simplest topologies for which these problems are  $\mathcal{NP}$ -hard.

We consider three different algorithms  $A_1$ ,  $A_2$ , and  $A_3$ . While  $A_1$  achieves a good approximation ratio only in the unweighted case,  $A_2$  and  $A_3$  produce solutions of guaranteed quality also in the weighted case.

In the remainder of this section, we give the required definitions and discuss previous work. In Sect. 2, we explain the three algorithms and show how they can be implemented efficiently. In Sect. 3, we report the experimental results that we obtained with our implementations.

### 1.1 Preliminaries

A bidirected tree is the directed graph obtained from an undirected tree by replacing each undirected edge by two directed edges with opposite directions.

An instance of the maximum weight edge-disjoint paths problem (MWEDP) in bidirected trees is given by a bidirected tree  $T = (V, E)$  and a set  $P$  of directed paths in  $T$ . Each path  $p$  is specified by its source node  $s_p$ , its destination node  $d_p$ , and a positive weight  $w_p$ . In the unweighted case (called MEDP), we assume that  $w_p = 1$  for all  $p \in P$ . A feasible solution is a subset  $P' \subseteq P$  of the given paths such that the paths in  $P'$  are edge-disjoint. We call the paths in  $P'$  the *accepted* paths. The goal is to maximize the total weight  $\sum_{p \in P'} w_p$  of the accepted paths. An algorithm for MWEDP is a  $\rho$ -approximation algorithm if it runs in polynomial time and always outputs a feasible solution whose total weight is at least a  $(1/\rho)$ -fraction of the optimum.

We say that two paths *intersect* if they share a directed edge of  $T$ . For a given set  $P$  of paths, the *load*  $L(e)$  of a directed edge  $e$  of  $T$  is the number of paths in  $P$  that use edge  $e$ , and  $L$  denotes the maximum load among all edges of  $T$ . A coloring of a given set of paths is an assignment of colors to the paths such that intersecting paths receive different colors. *Path coloring* is the problem of computing a coloring that minimizes the number of colors.

Let a bidirected tree  $T = (V, E)$  and a set  $P$  of directed paths in  $T$  be given. Take  $n = |V|$  and  $m = |P|$ . We assume that an arbitrary node  $r$  of  $T$  has been selected as the root of  $T$ . For  $v \neq r$ ,  $p(v)$  denotes the parent of  $v$ . For all  $v \in V$ , let  $d(v)$  denote the number of children of  $v$ . The *level* of a node is its distance (number of edges) from  $r$ . For a pair  $(u, w)$  of nodes in  $T$ , we denote by  $\text{lca}(u, w)$  the unique *least common ancestor* (lca) of  $u$  and  $w$ , i.e., the node with smallest level among all nodes on the path from  $u$  to  $w$ . For a path  $p$  from  $u$  to  $w$ , we take  $\text{lca}(p) = \text{lca}(u, w)$  and call  $\text{lca}(p)$  the lca of the path  $p$ . The one or two edges on a path  $p$  that are incident to  $\text{lca}(p)$  are called the *top edges* of  $p$ . A subtree of  $T$  *contains* a path  $p$  if  $\text{lca}(p)$  is a node of the subtree. By  $P_v$  we denote the subset of  $P$  that contains all paths  $p$  with  $\text{lca}(p) = v$ .

For a given set  $Q$  of paths with the same lca  $v$ , a subset  $Q'$  of edge-disjoint paths with maximum cardinality can be determined efficiently by computing a maximum matching in a bipartite graph  $G$ . The vertices of  $G = (\hat{V}, \hat{E})$  correspond to the incoming resp. outgoing edges of  $v$  in  $T$ , and the edges of  $G$  correspond to the paths in  $Q$ . A path in  $T$  with top edges  $x$  and  $y$  corresponds to an edge  $(x, y)$  of  $G$ .  $G$  has at most  $\min\{2d(v), 2|Q|\}$  non-isolated vertices and  $|Q|$  edges. A maximum matching in the bipartite graph  $G$  can be computed in time  $O(|\hat{V}|^{0.5} \cdot |\hat{E}|) = O(\sqrt{\min\{d(v), |Q|\}} \cdot |Q|)$  (see [12] and [13, Chapter 7.6]).

### 1.2 Related Work

MEDP in bidirected trees has first been studied in [7]. It was proved that MEDP is  $\mathcal{NP}$ -hard (in fact, MAX SNP-hard) for bidirected trees of arbitrary degree, but can be solved optimally in polynomial time for bidirected trees of constant degree. Furthermore, for every fixed  $\varepsilon > 0$ , a combinatorial  $(\frac{5}{3} + \varepsilon)$ -approximation algorithm for MEDP was presented.

MEDP and MWEDP can also be viewed as an independent set problem in the conflict graph of the given paths. For a different application of the weighted independent set problem, throughput maximization under real-time scheduling

**Table 1.** Approximation algorithms studied in this paper

$A_1$	combinatorial $(\frac{5}{3} + \varepsilon)$ -approximation for MEDP [7]
$A_2$	combinatorial 2-approximation for MWEDP (this paper, derived from [3])
$A_3$	LP-based $(\frac{5}{3} + \varepsilon)$ -approximation for MWEDP [8]

on multiple machines, a combinatorial two-pass algorithm that achieves approximation ratio 2 was devised in [3] (see also [1]). We will show in Sect. 2.2 that this approach can be adapted to our problem and yields a 2-approximation algorithm for MWEDP in bidirected trees.

In [8], it was shown that a general technique for converting coloring algorithms into maximum weight independent set algorithms, based on linear programming and inspired by the work in [2], can be used to obtain for every fixed  $\varepsilon > 0$  a  $(\frac{5}{3} + \varepsilon)$ -approximation algorithm for MWEDP in bidirected trees. This matches the approximation ratio of the best known algorithm for the unweighted case [7].

Path coloring is  $\mathcal{NP}$ -hard for bidirected trees (even in the case of binary trees), but there is an efficient approximation algorithm that uses at most  $5L/3$  colors for any set of paths with maximum load  $L$  [9, 10]. An efficient implementation with running-time  $O(nL^2)$ , where  $n$  is the number of nodes of  $T$ , was presented in [6]. The MWEDP algorithm from [8] uses this path coloring algorithm as a subroutine.

## 2 Approximation Algorithms for MEDP and MWEDP

We consider three approximation algorithms  $A_1$ ,  $A_2$  and  $A_3$  for MEDP and MWEDP in bidirected trees. Algorithm  $A_1$  is the  $(\frac{5}{3} + \varepsilon)$ -approximation algorithm for MEDP from [7]. Algorithm  $A_2$  is a 2-approximation algorithm for MWEDP that is obtained by adapting the approach of [3] to our problem. Finally, Algorithm  $A_3$  is the one obtained using linear programming and a path coloring subroutine as described in [8]. The characteristics of the three algorithms are summarized in Table 1.

We have implemented the algorithms in C++ using the LEDA library [13]. In particular, we use the LEDA function `MAX_CARD_BIPARTITE_MATCHING` to compute maximum matchings in bipartite graphs. For algorithm  $A_3$ , CPLEX [4] was used in addition. When an algorithm is called, it is passed the tree  $T$  as a LEDA graph and the set  $P$  as a list of paths, where each path is an object containing pointers to source node and destination node as well as an integer weight. The algorithm returns the list of accepted paths.

### 2.1 Combinatorial $(\frac{5}{3} + \varepsilon)$ -Approximation Algorithm $A_1$ for MEDP

The combinatorial  $(\frac{5}{3} + \varepsilon)$ -approximation algorithm  $A_1$  for MEDP was presented in [7]. We give a brief (simplified) outline of the basic ideas of the algorithm in order to be able to discuss its efficient implementation. For a more detailed description of  $A_1$ , see [7] or [5, Sect. 5.3].

*Outline of Algorithm  $A_1$ .* Algorithm  $A_1$  works in two passes. The main work is done in the first pass. The tree  $T$  is traversed in a bottom-up fashion and during the processing of a node  $v$  the set  $P_v$  of paths with lca  $v$  is handled. After a node is processed, the status of all paths contained in the subtree of that node is one of the following:

- *rejected* (not included in the solution),
- *accepted* (included in the solution),
- *deferred* (part of a group of two deferred paths, precisely one of which will be added to the solution in the second pass), or
- *unresolved* (will be turned into a rejected, accepted, or deferred path at a later time during the first pass).

Groups of deferred paths are easy to handle and can be treated essentially like accepted paths in the first pass. Unresolved paths can further be categorized into the following types of groups of unresolved paths: *undetermined paths* (individual unresolved paths), *groups of exclusive paths* (groups of two intersecting paths with different lcas), and *groups of 2-exclusive paths* (groups of four paths that contain two edge-disjoint paths). After a subtree has been processed, it contains at most one group of unresolved paths.

Assume that  $A_1$  is about to process a node  $v$  during the first pass. Let  $P'_v$  denote the subset of paths in  $P_v$  that do not intersect previously accepted paths. The basic idea of the algorithm is to determine a maximum cardinality subset  $S$  of edge-disjoint paths in  $P'_v$  and to accept the paths in  $S$  and to reject those in  $P_v \setminus S$ . Such a locally optimal set  $S$  can be computed efficiently by reduction to the maximum matching problem in a bipartite graph (cf. Sect. 1.1). If  $S$  contains only one or two paths, however, the algorithm is often forced to create a group of unresolved paths or a group of deferred paths, because its approximation ratio could not be better than 2 otherwise. For example, if  $S$  consists of a single path, accepting this path could block two paths with lcas of smaller level that might be accepted in an optimal solution instead.

The introduction of unresolved paths creates the new difficulty that, while processing  $v$ , the algorithm must consider not only the paths in  $P_v$ , but also the unresolved paths that are contained in subtrees under  $v$ . Let  $U_v$  denote the set of all unresolved paths in subtrees under  $v$  at the time  $v$  is processed. It is not known whether the computation of a maximum cardinality subset of edge-disjoint paths in  $P'_v \cup U_v$  can be carried out efficiently. The solution proposed in [7] is to try out all possible ways of accepting and rejecting paths in  $U_v$  if there are fewer than  $\frac{2}{\varepsilon}$  edge-disjoint paths in  $P'_v$  and fewer than  $\frac{2}{\varepsilon}$  subtrees with unresolved paths (there are at most  $4^{1/\varepsilon}$  meaningful possibilities to consider in this case), and to settle with an approximation of the locally optimal set  $S$  otherwise. The parameter  $\varepsilon$  can be an arbitrary positive constant and represents a trade-off between approximation ratio and running-time: algorithm  $A_1$  has approximation ratio bounded by  $\frac{5}{3} + \varepsilon$ , while the running-time is multiplied by  $4^{1/\varepsilon}$ . An outline of the processing of a node  $v$  during the first pass is shown in Fig. 1.

If there are any unresolved paths left after the root node is processed (at the end of the first pass), edge-disjoint paths among them are accepted greedily. The purpose of the second pass is then to select one of the two paths in each group of deferred paths. This can be implemented in a single tree traversal in top-down fashion.

1. Determine the set  $P'_v$  of paths in  $P_v$  that do not intersect previously accepted paths.
2. Case 1: There are less than  $\frac{2}{\varepsilon}$  subtrees with unresolved paths under  $v$  and less than  $\frac{2}{\varepsilon}$  edge-disjoint paths in  $P'_v$ .  
**for** each possibility  $\sigma$  of accepting/rejecting paths in  $U_v$  **do**  
     Let  $U_v(\sigma)$  be the set of (tentatively) accepted paths from  $U_v$ .  
     Determine a maximum cardinality set  $P'_v(\sigma)$  of edge-disjoint paths in  $P'_v$  not intersecting a path from  $U_v(\sigma)$ .  
     Let  $S$  be the set  $U_v(\sigma) \cup P'_v(\sigma)$  of maximum cardinality, over all possibilities  $\sigma$ . If  $|S| \geq 3$ , accept the paths in  $S$  and reject the paths in  $(P_v \cup U_v) \setminus S$ . If  $|S| < 3$ , compare the configuration of the paths in  $P'_v \cup U_v$  with a list of pre-specified configurations and handle the paths as instructed by the rules given for the configuration that applies. (The list of configurations and the rules for each configuration are the heart of algorithm  $A_1$ , see [7]. With the use of appropriate data structures, the right configuration can be determined in time  $O(|P'_v|)$ .)
3. Case 2: There are at least  $\frac{2}{\varepsilon}$  subtrees with unresolved paths under  $v$  or there are at least  $\frac{2}{\varepsilon}$  edge-disjoint paths in  $P'_v$ .  
     Calculate four candidate sets  $S_1, S_2, S_3$ , and  $S_4$  of edge-disjoint paths from  $P'_v \cup U_v$ . Two of the sets are computed by determining a maximum matching in a bipartite graph, and the other two sets are derived from the first two sets by performing a number of exchanges.  
     Let  $S_i$  be the set of maximum cardinality among  $S_1, S_2, S_3$ , and  $S_4$ .  
     Accept the paths in  $S_i$  and reject the paths in  $(P_v \cup U_v) \setminus S_i$ .

**Fig. 1.** Processing of  $v$  during the first pass of  $A_1$

*Efficient Implementation of Algorithm  $A_1$ .* We have implemented algorithm  $A_1$  as follows. As a first step, for each given path  $p \in P$  its lca as well as its top edges are computed in total time  $O(n+m)$ . For this purpose, we use the algorithm due to Schieber and Vishkin [14] that allows to answer lca queries for pairs of nodes in  $T$  in constant time after  $O(n)$  preprocessing. We modified the algorithm so that an lca query for nodes  $u$  and  $w$  does not only return the node  $\text{lca}(u, w)$ , but also the top edges of the path from  $u$  to  $w$ .

Next, for each  $v \in V$  the list  $P_v$  containing all paths in  $P$  with lca  $v$  is computed and stored at  $v$ . This is done in a single pass through the list  $P$ , taking time  $O(m)$ . A depth-first search (dfs) of  $T$  is used to compute in time  $O(n)$  for every node  $v$ : the level  $\text{level}(v)$ , the preorder number  $\text{pre}(v)$ , the maximum preorder number  $\text{max}(v)$  in the subtree of  $v$ , a reference to the edge  $(p(v), v)$ , a reference to the edge  $(v, p(v))$ , and the number  $d(v)$  of children of  $v$ .

Note that with the information computed and stored so far, it can be decided in constant time whether any two paths  $p$  and  $q$  intersect or not. If  $p$  and  $q$  have lcas with the same level, they intersect if and only if they have a common top edge. If the lca of  $p$  has a smaller level than the lca of  $q$ , then  $p$  and  $q$  intersect if and only if  $p$  contains a top edge of  $q$ , and this can be checked by determining whether the source node of  $p$  is contained in the subtree below the upward top edge of  $q$  or the destination of  $p$  is contained in the subtree below the downward top edge of  $q$ . (Checking whether a node  $u$  is contained in the subtree of a node  $w$  simply amounts to checking whether  $\text{pre}(w) \leq \text{pre}(u) \leq \text{max}(w)$ .)



During the first pass, we would also like to check in constant time whether a path  $p$  intersects *any* previously accepted path. For this purpose, we store with each edge  $e \in E$  a mark  $m(e)$ . Initially,  $m(e) = \text{false}$  for all  $e \in E$ . When a path  $q$  is accepted, we set  $m(e_1) = \text{true}$  for the upward top edge  $e_1$  of  $q$  and  $m(e) = \text{true}$  for all upward edges  $e$  in the subtree below  $e_1$ , and analogously for the downward top edge  $e_2$  of  $q$  and the downward edges in the subtree below  $e_2$ . Setting  $m(e) = \text{true}$  for an edge  $e$  is called *marking* the edge. All paths that are considered after  $q$  have lcas of smaller level, and they intersect  $q$  if and only if they intersect a top edge of  $q$ . It is easy to see that a path  $p$  intersects a previously accepted path if and only if the first edge on  $p$  or the last edge on  $p$  is marked, and this can be checked in constant time. For updating the values  $m(e)$  after a path  $q$  is accepted, it suffices to perform a dfs in the subtree below the upward top edge of  $q$  that marks all upward edges it encounters (and a similar dfs in the subtree below the downward top edge of  $q$ ). The dfs need not visit subtrees where the respective edges have already been marked. Therefore, the total time spent in marking edges throughout the algorithm is  $O(n)$ .

After a node  $w$  has been processed in the first pass, we store at  $w$  a reference  $U(w)$  to the group of unresolved paths contained in the subtree of  $w$ , if any. When a node  $v$  is processed, the values  $U(w)$  for all children  $w$  of  $v$  can be inspected to determine the number of subtrees with unresolved paths under  $v$ .

The processing of a node  $v$  is carried out as follows. First, the paths that intersect previously accepted paths are discarded from  $P_v$ , thus producing  $P'_v$ . The subtrees with unresolved paths can be counted in time  $O(d(v))$ , and a maximum number of edge-disjoint paths in  $P'_v$  can be computed in time  $O(\sqrt{\min\{d(v), |P'_v|\}} \cdot |P'_v|)$ . If Case 1 applies, all  $4^{1/\varepsilon}$  possible combinations  $\sigma$  of accepting paths  $U_v(\sigma) \subseteq U_v$  can be enumerated in constant time per combination. For each combination, we compute the subset  $Q \subseteq P'_v$  of paths in  $P'_v$  that do not intersect any path from  $U_v(\sigma)$ , in time  $O(|P'_v|)$ . Then the set  $P'_v(\sigma)$  can be obtained in time  $O(\sqrt{\min\{d(v), |Q|\}} \cdot |Q|)$  using a matching algorithm. After all combinations are computed, the locally optimal set  $S$  is known. If  $|S| \geq 3$ , the paths in  $S$  are accepted and all paths in  $(U_v \cup P_v) \setminus S$  are rejected. If  $|S| \leq 2$ , the applicable configuration is determined in time  $O(|P'_v|)$  and the corresponding rules (for example, creation of a new group of unresolved paths) are applied. Excluding the time for marking (which is accounted separately), the time for processing  $v$  can be bounded by  $O(d(v) + |P_v| + 4^{1/\varepsilon} \sqrt{\min\{d(v), |P_v|\}} \cdot |P_v|)$ .

If Case 2 applies, only the four candidate sets  $S_1, S_2, S_3$ , and  $S_4$  need to be computed. One set is obtained by taking a maximum cardinality subset of edge-disjoint paths in  $P'_v$  and then adding unresolved paths from  $U_v$  in a greedy manner. Another set is obtained by taking all undetermined paths from  $U_v$ , adding a maximum cardinality subset of edge-disjoint paths in  $P'_v$  that do not intersect any undetermined path, and then adding further unresolved paths in a greedy manner. The third and the fourth set are obtained by taking one of the previous two sets and then deleting paths from  $P'_v$  that are in the set so that a maximum number of exclusive paths from  $U_v$  can be added. Computing the first two sets is a matching problem, while the other two sets can be obtained from the first two sets in time  $O(d(v))$ . The paths in the set  $S_i$  of maximum cardinality are accepted, and all paths in  $(U_v \cup P_v) \setminus S_i$  are rejected. Excluding the time for marking, the time for processing  $v$  can be bounded by  $O(d(v) + |P_v| + \sqrt{\min\{d(v), |P_v|\}} \cdot |P_v|)$ .

The second pass of algorithm  $A_1$ , which is a top-down traversal of the tree in which one path is accepted from each group of deferred paths, can be implemented to run in time  $O(n)$  in a straightforward way.

Thus, the total running-time of our implementation of algorithm  $A_1$  can be given as  $\sum_{v \in V} O(d(v) + |P_v| + 4^{1/\varepsilon} \sqrt{\min\{d(v), |P_v|\}} \cdot |P_v|)$ . Using  $\sum_{v \in V} d(v) = n - 1$ ,  $\sum_{v \in V} |P_v| = m$ , and  $\sum_{v \in V} 4^{1/\varepsilon} \sqrt{\min\{d(v), |P_v|\}} \cdot |P_v| \leq 4^{1/\varepsilon} \min\{\sqrt{n} \cdot m, m^{1.5}\}$ , we obtain the upper bound  $O(n + m + 4^{1/\varepsilon} \min\{\sqrt{n} \cdot m, m^{1.5}\})$  on the total time.

Further optimizations (which we did not yet implement) can be derived from the observation that two paths in  $P_v$  are equivalent with respect to the calculations of  $A_1$  if they use the same top edges and if they intersect all unresolved paths in subtrees below  $v$  in exactly the same way. Therefore, only  $O(d(v)^2)$  paths in  $P_v$  are relevant for the calculations at  $v$ . If Case 1 applies, the number of relevant paths in  $P_v$  can even be bounded by  $O(d(v)/\varepsilon)$ . Thus the processing of  $v$  takes time  $O(d(v) + |P_v| + \sqrt{\min\{d(v), |P_v|\}} \cdot \min\{|P_v|, d(v)^2\} + 4^{1/\varepsilon} \sqrt{\min\{d(v), |P_v|\}} \cdot \min\{|P_v|, d(v)/\varepsilon, d(v)^2\})$ , allowing us to bound the total running-time of  $A_1$  by:

$$O(m + n + \min\{\sqrt{n} \cdot m, n^{2.5}, m^{1.5}\} + 4^{1/\varepsilon} \min\{\sqrt{n} \cdot m, n^{2.5}, m^{1.5}, \frac{1}{\varepsilon} n^{1.5}\})$$

## 2.2 Combinatorial 2-Approximation Algorithm $A_2$ for MWEDP

Algorithm  $A_2$  is shown in Fig. 2. It works in two passes as well. In the first pass, it processes the given paths  $p \in P$  in order of nonincreasing levels of their *lcas* and maintains a stack  $S$  of paths with *values*. The value  $v_p$  of a path  $p$  is not necessarily equal to the weight  $w_p$  of that path. Initially,  $S$  is empty. At any time, let  $val(S') = \sum_{q \in S'} v_q$  for any  $S' \subseteq S$ . When  $A_2$  processes a path  $p$  during the first pass, let  $S_p$  be the set of all paths in  $S$  that intersect  $p$ . If  $w_p \leq val(S_p)$ , the algorithm does nothing. If  $w_p > val(S_p)$ , it pushes  $p$  onto the stack and assigns it the value  $w_p - val(S_p)$ . In the second pass, the paths are popped from the stack one by one, and each path is accepted if it does not intersect any previously accepted path.

If  $A_2$  is applied to an unweighted instance of MEDP, it corresponds to a pure greedy algorithm that processes the paths in order of nonincreasing levels of their *lcas*.

First, we claim that  $2val(S) \geq OPT$  holds at the end of the first pass, where  $OPT$  is the total weight of an optimal solution. For any path  $p$  in an optimal solution, the total value of all paths that intersect  $p$  and that are in  $S$  right after  $p$  is processed is at least  $w_p$  (by definition of  $A_2$ ). Furthermore, every path  $q$  in  $S$  intersects at most two paths that are in the optimal solution and that are processed after  $q$ , because each such path intersects at least one of the two top edges of  $q$ . This shows  $2val(S) \geq OPT$ .

Further, the total weight of the paths that are accepted by  $A_2$  is at least  $val(S)$ , because a path  $p$  that is accepted in the second pass adds  $w_p$  to the total weight of the solution, while it reduces the total value of the remaining paths in  $S$  that do not intersect an accepted path by at most  $w_p$ . This is because  $val(S_p \cup \{p\})$  was exactly  $w_p$  at the time when  $p$  was pushed onto  $S$ . Therefore,  $A_2$  is a 2-approximation algorithm.

```

{ Phase 1 }
S ← empty stack ;
for all paths p in order of nonincreasing levels of their lcas do
    val ←  $\sum_{q \in S: q \text{ intersects } p} v_q$ ;
    if  $w_p > val$  then
         $v_p \leftarrow w_p - val$ ;
        push(S, p);
    fi;
od;
{ Phase 2 }
A ←  $\emptyset$ ; { set of accepted paths }
while S is not empty do
    p ← pop(S);
    if  $A \cup \{p\}$  is a set of edge-disjoint paths then  $A \leftarrow A \cup \{p\}$  fi;
od;
output A;

```

Fig. 2. Algorithm  $A_2$ 

The implementation of algorithm  $A_2$  is straightforward. First, a preprocessing in  $O(n + m)$  time is performed similar as for  $A_1$  (computing lcas of all paths and information to answer path intersection queries in constant time). When a path  $p$  is processed in the first pass,  $val(S_p)$  can be determined in time  $O(|S|)$  by checking intersection with all paths that are currently on the stack. Hence, the total running-time of the first pass can be bounded by  $O(n + m^2)$ . The second pass takes time  $O(n + |S|) = O(n + m)$ , because each path  $p$  that is popped from the stack intersects a previously accepted path if and only if a top edge of  $p$  is used by a previously accepted path, and this can be checked in time  $O(1)$  using an appropriate edge marking scheme. So the worst-case running time of  $A_2$  is bounded by  $O(n + m^2)$ . It can be expected to be better on average since the size of the stack will usually be much smaller than  $m$ . Furthermore, it is possible to improve the running-time  $O(|S|)$  for the computation of  $val(S_p)$  by using additional data structures, but we did not yet pursue this direction further.

### 2.3 LP-Based $(\frac{5}{3} + \varepsilon)$ -Approximation Algorithm $A_3$ for MWEDP

Consider the following linear programming relaxation, which uses a variable  $x_p$  for each path  $p \in P$  that indicates whether  $p$  is accepted ( $x_p = 1$ ) or rejected ( $x_p = 0$ ):

$$\begin{aligned}
 & \max \sum_{p \in P} w_p x_p & (1) \\
 & \text{s.t. } \sum_{p: e \in p} x_p \leq 1, \forall e \in E \\
 & \quad 0 \leq x_p \leq 1, \forall p \in P
 \end{aligned}$$

If we impose integrality conditions on the variables  $x_p$ , (1) becomes an integer linear program that models MWEDP exactly. With the relaxed conditions  $0 \leq$

$x_p \leq 1$ , (1) is a linear program that can be solved optimally in polynomial time. Let  $\hat{x}$  denote an optimal solution vector of (1) and let  $\hat{z} = \sum_{p \in P} w_p \hat{x}_p$  denote the corresponding value of the objective function. Algorithm  $A_3$  rejects all paths  $p$  with  $\hat{x}_p = 0$ , accepts all paths with  $\hat{x}_p = 1$ , and deals with the remaining paths (those with  $0 < \hat{x}_p < 1$ ) as follows. (For ease of discussion, assume that all  $p \in P$  satisfy  $\hat{x}_p < 1$ .) First, it rounds these  $\hat{x}_p$  to multiples of  $1/N$  for some integer  $N$  such that the resulting vector  $\bar{x}$  is still a feasible solution to (1) and such that the objective value  $\bar{z}$  corresponding to  $\bar{x}$  is at least  $(1 - \delta)\hat{z}$ , where  $\delta = \varepsilon/(\frac{5}{3} + \varepsilon)$ .  $N$  can be chosen as  $\lceil 4m/\delta^2 \rceil$ , as shown in [8]. In our implementation, we found that  $N = 30$  was usually sufficient for achieving  $\bar{z} \geq (1 - \delta)\hat{z}$ . A set  $Q$  of paths is then created by starting with the empty set and adding  $k_p$  copies of every path  $p \in P$  provided that  $\bar{x}_p = k_p/N$ . As  $\bar{x}$  is a feasible solution to (1),  $Q$  has maximum load at most  $N$ . Next, the path coloring algorithm from [10, 6] is used to partition  $Q$  into at most  $5N/3$  color classes (sets of edge-disjoint paths). By the pigeonhole principle, at least one of the color classes must have a total weight that is at least  $\frac{3}{5}\bar{z}$ . Algorithm  $A_3$  accepts the paths in the color class with maximum weight (in addition to the paths that were accepted because we had  $\hat{x}_p = 1$  right from the start).

We used CPLEX as a solver for the linear program (1). The LP has  $m$  variables and  $O(n)$  constraints, and it is created and passed to the CPLEX Callable Library in  $O(mn)$  time. After the LP solution is obtained, the rounding and the creation of the coloring instance are done in time  $O(m)$ . The implementation of the path coloring algorithm, which was taken from [6], has running time  $O(nL^2)$  if the maximum load is  $L$ . As noted above, we have  $L \leq N$  in our case. So the total running-time of algorithm  $A_3$  is  $O(mn + nN^2)$  plus the time spent in solving the LP. The LP has a special structure (it is a fractional packing problem) and is usually solved very quickly by CPLEX.

CPLEX can also be used to solve the ILP version of (1), i.e., to enforce  $x_p \in \{0, 1\}$  for all  $p \in P$ . The running-time for obtaining the optimal ILP solution can, of course, be exceedingly long. Nevertheless, we obtained optimal solutions quickly for many instances in this way.

### 3 Experimental Results

For the algorithms discussed in the previous section, we are interested in the running-time and in the number (weight) of accepted paths (as compared to an optimal solution) on instances of MEDP and MWEDP that are generated randomly by various different methods. All experiments were run on a SUN Ultra 10 workstation.

The following tree topologies are considered: complete (balanced)  $k$ -ary trees of depth  $d$  (called  $\text{bal}(k, d)$  and having  $n = (k^{d+1} - 1)/(k - 1)$  nodes), caterpillars  $\text{cat}(\ell, d)$  with a backbone (spine) of length  $\ell + 1$  and  $d$  spikes (nodes of degree one) adjacent to each backbone node except the last one (therefore,  $n = \ell(d + 1) + 1$ ), and flat trees  $\text{flat}(d)$  consisting of a root node  $v$  with degree  $d$  such that each child of  $v$  is the root of a complete binary tree of depth 2 ( $n = 1 + 7d$ ). Sets of paths in the trees were generated by choosing source and destination randomly among all nodes (this method is indicated by [A] in the tables) or only among all leaves of the tree (indicated by [L]). In the weighted case, the weights of the paths were either selected uniformly at random from the integers between 1 and

100 or set equal to  $5\ell - 4$  for paths of length  $\ell$  (thus making long paths more valuable than short paths).

We also constructed instances of MEDP following the reduction that was used to prove MEDP  $\mathcal{NP}$ -hard for bidirected trees [7, 11]. An instance of 3-DIMENSIONAL MATCHING (3DM) is given by three disjoint sets  $X$ ,  $Y$  and  $Z$  of cardinality  $h$  and a set  $M$  of triples from  $X \times Y \times Z$ . It is  $\mathcal{NP}$ -complete to decide whether there exist  $h$  disjoint triples in  $M$ . For given  $h$  and  $k$ , we first construct an instance of 3DM by setting  $|X| = |Y| = |Z| = h$  and generating  $hk$  random triples such that each element of  $X$ ,  $Y$ , or  $Z$  occurs in exactly  $k$  triples. From this instance of 3DM, a tree with  $n = 1 + 3h + 2hk$  nodes and  $3hk$  paths in the tree are derived. The optimal solution of the resulting instance  $\text{tdm}(h, k)$  of MEDP lies somewhere between  $hk$  and  $h(k + 1)$ , while algorithms  $A_1$  and  $A_2$  always accept exactly  $hk$  paths.

First, we discuss our results for unweighted paths. Table 2 lists the number of accepted paths and the running-times of the algorithms (approximation algorithms  $A_1$ ,  $A_2$ , and  $A_3$  as well as the optimal algorithm calling the ILP solver of CPLEX) on a number of instances generated randomly as described above. (For example, the row “bal(3, 7), 500[A]” shows the results for a balanced 3-ary tree of depth 7 with 500 paths generated by choosing pairs of arbitrary nodes randomly.) In addition, the optimum  $\hat{z}$  of the LP relaxation (1) and the number of fractional paths (i.e., paths  $p$  with  $0 < \hat{x}_p < 1$ ) in the corresponding solution (computed by CPLEX as part of  $A_3$ ) are given. The parameter  $\varepsilon$  for  $A_1$  and  $A_3$  was set to 0.1. (Choosing smaller values of  $\varepsilon$  did not improve the results significantly, as  $A_1$  and  $A_3$  achieved approximation ratios substantially better than the worst-case bound of  $\frac{5}{3} + \varepsilon$  in our experiments, anyway.) Each experiment was repeated ten times, and the values in Table 2 give the average of the ten measurements.

Surprisingly, it turned out that all instances with randomly generated paths are in fact very “easy,” even in trees with vertices of high degree: the solutions computed by the approximation algorithms (in particular, by  $A_1$  and  $A_3$ ) are very close to the optimal solution, and even the optimal solution is found very quickly by CPLEX. We observed that the number of variables with fractional values in the LP solution was usually very small or zero. This means that the LP solution is almost integral already, making it plausible that the ILP can be solved quickly. Interestingly, the ILP solver took longer than the LP solver even when the LP solution was integral.

With the instances  $\text{tdm}(h, k)$ , the results were substantially different. Here, the time requirement for solving the ILP becomes exceedingly large even for moderate values of  $h$  and  $k$ . For example, instances  $\text{tdm}(40, 5)$  took around one hour to solve optimally, while instances  $\text{tdm}(50, 8)$  required more than 15 hours.

Summarizing the results for MEDP, we can say that  $A_1$  is a very fast algorithm that produces excellent solutions for randomly generated sets of paths. Algorithm  $A_3$  is less efficient, but produces better solutions on “worst-case” inputs  $\text{tdm}(h, k)$ . Algorithm  $A_1$  is preferable to  $A_2$  for MEDP.

The experimental results for MWEDP are shown in Table 3.  $A_2$  and  $A_3$  are compared with the optimal solution and the LP relaxation. The total weight of one solution (the optimal solution, where available) is given in absolute value, while for the other solutions the relative deviation is shown. Instances  $\text{tdm}'(h, k)$  are created like  $\text{tdm}(h, k)$ , but the paths are assigned random weights in  $\{1, 2, \dots, 100\}$ . Again, it turned out that MWEDP in bidirected trees is an easy problem

**Table 2.** Experimental results for MEDP (average of ten runs)

MEDP instance	# accepted paths				LP		running-time (cpu secs.)			
	$A_1$	$A_2$	$A_3$	OPT	$\hat{z}$	#fract.	$A_1$	$A_2$	$A_3$	OPT
bal(3, 7),500[A]	39.8	39.9	41.7	41.8	42.1	8.5	0.11	0.10	0.29	0.75
bal(3, 7),500[L]	38.1	38.1	39.4	39.6	39.6	7.0	0.11	0.10	0.25	0.73
bal(50, 2),4000[A]	128.1	126.6	128.1	128.1	128.1	0.0	0.26	0.84	0.35	0.70
bal(20, 3),8000[L]	251.1	235.0	251.1	251.1	251.1	0.0	0.43	3.68	1.00	1.80
cat(2000, 2),2000[A]	69.6	69.6	69.6	69.6	69.6	0.0	0.22	0.32	10.10	29.30
cat(1000, 9),2000[A]	71.8	71.8	71.9	71.9	71.9	0.0	0.36	0.45	5.62	18.91
flat(500),3000[A]	502.2	431.8	502.2	502.2	502.2	0.0	0.28	1.01	0.68	1.37
flat(500),3000[L]	502.5	433.3	502.5	502.5	502.5	0.0	0.39	1.01	0.83	1.75
tdm(20, 4)	80.0	80.0	94.8	99	99.9	164.3	0.02	0.04	0.58	2.97
tdm(30, 4)	120.0	120.0	140.2	149.0	150.0	259.8	0.03	0.08	0.93	18.00
tdm(40, 5)	200.0	200.0	225.6	—	240.0	342.8	0.05	0.20	1.35	—
tdm(50, 8)	400.0	400.0	433.8	—	450.0	423.7	0.07	0.73	2.15	—
tdm(200, 10)	2000.0	2000.0	2122.5	—	2200.0	1704.6	0.37	17.90	36.61	—

**Table 3.** Experimental results for MWEDP (average of ten runs)

MWEDP instance	weight of accepted paths			LP		running-time			
	$A_2$	$A_3$	OPT	$\hat{z}$	#fract.	$A_2$	$A_3$	OPT	
path weights chosen randomly from $\{1, 2, \dots, 100\}$ :									
bal(2, 7),500[A]	-3.17%	-0.55%	2244.1	+0.14%	5.5	0.05	0.10	0.29	
bal(3, 7),500[A]	-5.97%	-0.28%	2446.9	+0.05%	7.5	0.11	0.23	0.57	
bal(3, 7),500[L]	-4.67%	-0.46%	2463.5	+0.02%	2.1	0.11	0.20	0.55	
bal(50, 2),4000[A]	-10.99%	-0.00%	8641.4	+0.00%	0.0	1.04	0.39	0.48	
bal(20, 3),8000[L]	-4.16%	-0.00%	15089.0	+0.00%	0.0	4.90	1.53	1.43	
cat(2000, 2),2000[A]	-0.08%	-0.00%	4247.4	+0.00%	0.0	0.40	9.90	44.13	
cat(1000, 9),2000[A]	-0.05%	-0.00%	4558.6	+0.00%	0.0	0.54	5.34	29.32	
flat(500),3000[A]	-14.88%	-0.00%	37543.4	+0.00%	0.0	1.63	0.49	0.65	
flat(500),3000[L]	-14.34%	-0.00%	37441.1	+0.00%	0.0	1.65	0.55	0.76	
tdm'(20, 4)	-4.63%	-0.19%	5703.6	+0.01%	10.5	0.04	0.11	0.07	
tdm'(30, 4)	-4.14%	-0.20%	8432.1	+0.01%	9.9	0.08	0.20	0.11	
tdm'(40, 5)	-3.47%	-0.14%	13599.1	+0.02%	20.9	0.21	0.27	0.59	
tdm'(50, 8)	-3.26%	-0.25%	24981.0	+0.03%	102.6	0.77	0.95	7.77	
tdm'(200, 10)	-2.78%	-0.36%	—	121435.1	589.4	19.01	26.92	—	
path weights chosen as $5\ell - 4$ for paths of length $\ell$ :									
bal(2, 7),500[A]	-9.37%	-2.39%	881.7	+0.49%	24.4	0.05	0.21	0.64	
bal(3, 7),500[A]	-6.36%	-1.25%	1652.0	+0.36%	13.4	0.12	0.34	0.71	
bal(3, 7),500[L]	-5.18%	-0.00%	1714.6	+0.00%	0.0	0.11	0.19	0.65	
bal(50, 2),4000[A]	-1.43%	-0.00%	1250.1	+0.00%	0.0	0.88	0.35	0.73	
bal(20, 3),8000[L]	-7.22%	-0.00%	4035.6	+0.00%	0.0	3.87	1.12	1.83	
cat(2000, 2),2000[A]	-0.01%	-0.00%	19900.7	+0.00%	0.0	1.21	11.98	32.49	
cat(1000, 9),2000[A]	-0.02%	-0.00%	10004.2	+0.00%	0.0	1.38	7.16	21.00	
flat(500),3000[A]	-17.42%	-0.00%	12019.9	+0.00%	0.0	1.42	0.75	1.55	
flat(500),3000[L]	-13.74%	-0.00%	13005.0	+0.00%	0.0	1.03	0.87	1.79	

if the paths are generated randomly. The solution produced by  $A_3$  was always within 1 percent of the optimum, and in many cases the solution of the LP relaxation was already integral. In the caterpillar trees,  $A_2$  was substantially faster than  $A_3$  and produced solutions of comparable quality. On the other inputs, the solutions obtained with  $A_3$  were consistently better than those of  $A_2$ .

It is an interesting open problem to find families of instances (other than  $\text{tdm}(h, k)$ ) where it is more difficult to compute solutions that are optimal or close to optimal.

## References

1. A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. S. Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing STOC'00*, pages 735–744, 2000.
2. A. Bar-Noy, S. Guha, J. S. Naor, and B. Schieber. Approximating the throughput of multiple machines under real-time scheduling. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing STOC'99*, pages 622–631, 1999.
3. P. Berman and B. DasGupta. Improvements in throughput maximization for real-time scheduling. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing STOC'00*, pages 680–687, 2000.
4. CPLEX 6.5, 1999. <http://www.cplex.com/>.
5. T. Erlebach. *Scheduling Connections in Fast Networks*. PhD thesis, Technische Universität München, 1999.  
<http://www.tik.ee.ethz.ch/~erlebach/dissertation.ps.gz>.
6. T. Erlebach and K. Jansen. Efficient implementation of an optimal greedy algorithm for wavelength assignment in directed tree networks. In K. Mehlhorn, editor, *Proceedings of the 2nd Workshop on Algorithm Engineering WAE'98*, Technical Report MPI-I-98-1-019, pages 13–24, Max-Planck-Institut für Informatik, Saarbrücken, August 1998.
7. T. Erlebach and K. Jansen. Maximizing the number of connections in optical tree networks. In *Proceedings of the 9th Annual International Symposium on Algorithms and Computation ISAAC'98*, LNCS 1533, pages 179–188, 1998.
8. T. Erlebach and K. Jansen. Conversion of coloring algorithms into maximum weight independent set algorithms. In *ICALP Workshops 2000*, Proceedings in Informatics 8, pages 135–145. Carleton Scientific, 2000.
9. T. Erlebach, K. Jansen, C. Kaklamanis, M. Mihail, and P. Persiano. Optimal wavelength routing on directed fiber trees. *Theoretical Computer Science*, 221:119–137, 1999. Special issue of ICALP'97.
10. T. Erlebach, K. Jansen, C. Kaklamanis, and P. Persiano. An optimal greedy algorithm for wavelength allocation in directed tree networks. In *Proceedings of the DIMACS Workshop on Network Design: Connectivity and Facilities Location*, volume 40 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 117–129. AMS, 1998.
11. N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees, with applications to matching and set cover. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming ICALP'93*, LNCS 700, pages 64–75, 1993.
12. J. Hopcroft and R. Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.
13. K. Mehlhorn and S. Näher. *LEDA — A Platform for Combinatorial and Geometrical Computing*. Cambridge University Press, 1999.
14. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.

# Dynamic Maintenance Versus Swapping: An Experimental Study on Shortest Paths Trees<sup>\*</sup>

Guido Proietti<sup>1,2</sup>

<sup>1</sup> Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, Via  
Vetoio, 67010 L'Aquila, Italy

proietti@univaq.it.

<sup>2</sup> Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche,  
Viale Manzoni 30, 00185 Roma, Italy

**Abstract.** Given a spanning tree  $T$  of a 2-edge connected, weighted graph  $G$ , a *swap edge* for a failing edge  $e$  in  $T$  is an edge  $e'$  of  $G$  reconnecting the two subtrees of  $T$  created by the removal of  $e$ . A *best swap edge* is a swap edge enjoying the additional property of optimizing the swap, with respect to a given objective function.

If the spanning tree is a single source shortest paths tree rooted in a node  $r$ , say  $S(r)$ , it has been shown that there exist efficient algorithms for finding a best swap edge, for each edge  $e$  in  $S(r)$  and with respect to several objective functions. These algorithms are efficient both in terms of the functionalities of the trees obtained as a consequence of the swaps, and of the time spent to compute them. In this paper we propose an extensive experimental analysis of the above algorithms, showing that their actual behaviour is much better than what it was expected from the theoretical analysis.

## 1 Introduction

Let  $V$  be a set of  $n$  *sites* that must be interconnected, let  $E$  be a set of  $m$  potential *links* between the sites, and let  $w(e)$  be some nonnegative real *weight* associated with link  $e$ , expressing some property associated with  $e$ , like for instance a set-up cost or the distance between the two connected sites. Let  $G = (V, E)$  be the corresponding weighted, undirected graph, and assume that  $G$  is 2-edge connected (i.e., to disconnect  $G$  we have to remove at least 2 edges). Let  $G' = (V, E')$ ,  $E' \subseteq E$  be a connected spanning subgraph of  $G$ , allowing to all the nodes to communicate. We call  $G'$  a *communication network*, or simply a network, in  $G$ .

A network is generally built with the aim of minimizing some *cost*  $C(G')$ , which is computed using some criteria defined according to the weights of the edges in  $G'$ . For instance, if the cost corresponds to the sum of the weights of the network edges, that is  $C(G') = \sum_{e \in E'} w(e)$ , then the cheapest network coincides with a *minimum (weight) spanning tree* (MST) of  $G$ .

<sup>\*</sup> This work has been partially supported by the EU TMR Grant CHOROCHRONOS.



Apart from minimizing the network cost, there is also another important feature of the network that we must take care of, that is its ability to survive to individual network components faults, like for instance an edge corruption. This is called the *survivability* of the network [6]. In the past few years, survivability problems have been studied for several network topologies [1,2,9,11,15].

Clearly, low-cost and high-reliability of the network are two conflicting parameters: the network survivability can be strengthened by increasing its connectivity, but this will augment the network cost as well, though. In the extreme, to maintain costs as low as possible, a network might be designed as a spanning tree of the underlying graph. Such a network, however, will not even survive to a single edge failure. Hence, some redundancy in the edges must be introduced. Assuming that damaged edges can be restored quickly, the likelihood of having overlapping failures is quite small. Therefore, it makes sense to strengthen the survivability of such a network by dealing with the failure of each and every single edge in the network, since we can expect that sooner or later each edge will fail, thus providing for each edge a set of *replacement* (or *swap*) edges, that will maintain the network connected as a consequence of an edge failure.

Since the network cost is always in our mind, the set of swap edges should be chosen in such a way that the *replacement network*  $G'_{e/E_e}$ , as obtained after replacing an edge  $e$  with a set of edges  $E_e \subseteq E \setminus \{e\}$ , is still cheap in terms of the considered cost. On the other hand, the number of replacing edges should be small, to maintain set-up costs as low as possible. In tree-based networks, to satisfy both the above requirements, one could associate with each edge  $e$  in the network a single replacement edge  $e^*$ , namely a *best swap edge*, so that the resulting replacement network  $G'_{e/e^*}$  (named *best swap network*) will be the best possible among all the networks that can be obtained by means of a single replacement. Consequently, a *swap problem* asks for finding a best swap edge for every edge in the network.

Many network architectures are based on a *single source shortest paths tree* (SPT)  $S(r) = (V, E_S)$  rooted at a given node  $r$ . Let  $S_{e/e'}(r) = (V, E_S \cup \{e'\} \setminus \{e\})$  denote the SPT obtained by swapping  $e$  with  $e'$ , also named *swap tree*. For a SPT, the choice of a best swap edge depends on which tree functionality we are interested in maintaining as good as possible: In fact, we could be interested in choosing a swap edge minimizing the maximum distance of node  $r$  to any node disconnected from  $r$  after the failure, or alternatively minimizing the average distance, or whatever else is of interest from a network management point of view. Therefore, an exhaustive approach to the problem requires a rigorous definition of the *swap function* to be minimized. More precisely, let  $F$  be a function defined over all the possible swap trees with respect to a failing edge  $e$ . A best swap can be defined as a swap edge  $e^*$  minimizing  $F$ . Hence, a best swap tree will be denoted as  $S_{e/e^*}(r)$ , and the complexity of finding a best swap for any edge in  $S(r)$  will depend on the function  $F$ .

In a previous paper [12], we have shown that there exist efficient algorithms for finding a best swap edge, for each edge  $e \in E_S$ , and with respect to several swap functions. These algorithms are called *swap algorithms*, and are efficient for a two-fold reason: First, a best swap tree guarantees a functionality – as

measured with respect to the objective function leading the swap – analogous to a new real SPT (which in general requires a costly set-up of many new edges), and second, finding all the best swap trees is computationally much faster than computing all the new real SPTs.

In this paper we propose an extensive experimental analysis of the above algorithms, both in terms of the functionalities of the best swap trees, and of the time spent to compute them, showing that their actual behaviour is much better than what it was expected from the theoretical analysis. To test the tree functionalities, we take into consideration three features of primary importance in SPTs, showing that, in the average, the ratio of the features, as computed in a best swap tree and in a real new SPT, is maintained within a small constant factor, very close to 1. Astonishingly, this result is achieved with respect to all the above measures, *independently* of the function a best swap tree minimizes.

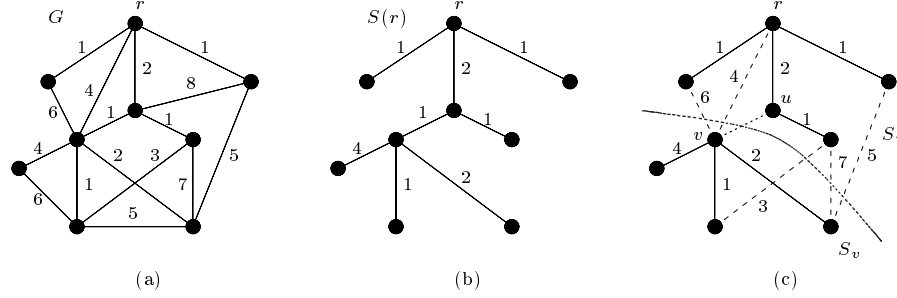
On the other hand, to assess experimentally the computational efficiency of swapping versus dynamic maintenance, we first show that swap algorithms are generally faster than recomputing all the new SPTs by means of the classic dynamic algorithms of Ramalingam and Reps [13], which has been shown to be very efficient [5]. Second, and more important, we show that the expected number of new edges entering in a new real SPT is a substantial fraction (about a half) of the number of nodes disconnected from the root after the edge failure, thus implying a high set-up cost for a new real SPT. Hence, we conclude that swapping is good and fast at the same time.

The paper is organized as follows. In Section 2, we define more precisely and formally the problems we are dealing with, and we briefly recall the results presented in [12], giving the theoretical worst case time complexity of the swap algorithms, along with the upper bounds of efficiency ratios. In Section 3, we present and comment our experiments, while finally, in Section 4, we list some open problems raising from the obtained results.

## 2 Some Interesting Swap Problems for SPTs

Let us first recall some of the basic definitions concerning a SPT that we will use in the sequel; for details on graph terminology, see [7].

Let  $G = (V, E)$  be a 2-edge connected, undirected graph, with  $n$  nodes and  $m$  edges having a nonnegative real length  $w(e)$  associated. A *simple path* (or a *path* for short) in  $G$  between two nodes  $v, v' \in V$  is a subgraph  $P(v, v')$  of  $G$  with node set  $V(P) = \{v \equiv v_1, v_2, \dots, v' \equiv v_k \mid v_i \neq v_j \text{ for } i \neq j\}$  and edge set  $E(P) = \{(v_i, v_{i+1}) \mid 1 \leq i < k\}$ . Path  $P(v, v')$  is said to go from  $v$  to  $v'$  and to have *length*  $|P(v, v')| = \sum_{e \in E(P)} w(e)$ . Among all the paths between two nodes, a path is called *shortest* if the sum of the lengths of the path edges is smallest. Let the *distance*  $d(v, v')$  between two nodes  $v, v'$  in  $G$  be the length of a shortest path between  $v$  and  $v'$ . For a distinguished node  $r \in V$ , called the *source*, and all the nodes  $v \in V \setminus \{r\}$ , a *shortest paths tree* (SPT)  $S(r) = (V, E_S)$  rooted in  $r$  is a spanning tree of  $G$  formed by the union of shortest paths, with one shortest path from  $r$  to  $v$  for each  $v \in V \setminus \{r\}$ . A *swap edge* for an edge  $e = (u, v) \in E_S$ , with  $u$  closer to  $r$  than  $v$ , is an edge  $e' = (u', v') \in E \setminus \{e\}$  reconnecting the two



**Fig. 1.** (a) A weighted graph  $G = (V, E)$ ; (b) a SPT  $S(r)$  rooted in  $r$ ; (c) edge  $e = (u, v)$  is removed from  $S(r)$ : dashed edges are swap edges.

subtrees of  $S(r)$  created by the removal of  $e$ , that is the subtree  $S_u$  containing  $r$  and  $u$ , and the subtree  $S_v$  containing  $v$ . Figure 1 illustrates a SPT  $S(r)$  of a graph  $G$ , along with the set of swap edges for a given edge  $e = (u, v) \in E_S$ .

The tree  $S_{e/e'}(r) = (V, E_S \cup \{e'\} \setminus \{e\})$ , obtained as a consequence of the swapping between  $e$  and  $e'$ , is named *swap tree*. In the following,  $d_{e/e'}(v, v')$  will denote the distance between  $v$  and  $v'$  in  $S_{e/e'}(r)$ . Let  $F$  be a function defined over all the possible swap trees with respect to a failing edge  $e$ . A *best swap edge* is a swap edge  $e^*$  minimizing  $F$ . Hence, the swap tree  $S_{e/e^*}(r)$  will be named a *best swap tree*.

Many network architectures are based on a SPT, especially centralized networks, where a source node broadcasts messages to all the other nodes. A SPT satisfies several properties, like for instance it minimizes the average distance from the root to any node, or it minimizes the *radius* (and the *diameter*) among all the spanning trees rooted in  $r$  (hence, if the root is the *center* of the underlying graph, it coincides with a minimum diameter spanning tree [8]).

Starting from these properties, which in some sense express the main features of a SPT, the following set of swap functions have been suggested in [12] for leading the swapping:

1.  $F_{\{r, \Sigma\}}(S_{e/e'}(r)) = \sum_{t \in S_v} d_{e/e'}(r, t);$
2.  $F_{\{r, \Delta\}}(S_{e/e'}(r)) = \max_{t \in S_v} \{d_{e/e'}(r, t) - d(r, t)\};$
3.  $F_{\{r, \max\}}(S_{e/e'}(r)) = \max_{t \in S_v} \{d_{e/e'}(r, t)\}.$

These functions focus on the distance between  $r$  and nodes in  $S_v$ , since it is interesting to study how the nodes disconnected from the root are affected by the swapping. Notice that minimizing these swap functions is equivalent to find a swap edge minimizing, respectively: (1) the average distance from the root  $r$  of any node in  $S_v$ ; (2) the radius of  $S_{e/e'}(r)$  restricted to  $S_v$ , and (3) the maximum increase of the distance from the root  $r$  to any node in  $S_v$ . The swap problems induced by these swap functions have been named the *Sum-Problem*,

**Table 1.** Time and space complexity and ratios for the studied swap algorithms, where  $\alpha(m, n)$  denote the functional inverse of the Ackermann's function [14].

Measure	Algorithm		
	Swap-Sum	Swap-Increase	Swap-Height
<i>Time</i>	$O(n^2)$	$O(m \cdot \alpha(m, n))$	$O(n\sqrt{m})$
<i>Space</i>	$O(n^2)$	$O(m)$	$O(m)$
$\rho_\Sigma$	3	3	unbounded
$\rho_\Delta$	unbounded	1	unbounded
$\rho_{\max}$	4	2	2

the *Increase-Problem* and the *Height-Problem*, respectively. Correspondingly, efficient swap algorithms for solving these problems have been devised, named **Swap-Sum**, **Swap-Increase** and **Swap-Height**, respectively, and having in the worst case time and space complexity as reported in Table 1. Note that these bounds positively compare with the fastest known dynamic solution for recomputing all the new SPTs, which costs  $O(nm + n^2 \log n)$  time and  $O(m)$  space [4,13]. Since swapping a single edge for a failed one is fast and involves very few changes in the underlying network (e.g., as to routing information), it is interesting to see how a best swap tree compares with a corresponding new real SPT  $S'_e(r)$  of  $G - e$ . While it is natural to study each of the three quality criteria for the algorithms that optimize the corresponding swap, the authors studied in [12] also the effect that a swap algorithm has on the other criteria (that it does not aim at). Let  $d'_e(v, v')$  denote the distance between  $v$  and  $v'$  in  $S'_e(r)$ . For each swap algorithm, the following ratios in the two trees have been studied:

$$\rho_\Sigma = \frac{\sum_{t \in V(S_v)} d_{e/e^*}(r, t)}{\sum_{t \in V(S_v)} d'_e(r, t)}; \quad \rho_\Delta = \frac{\max_{t \in S_v} \{d_{e/e^*}(r, t) - d(r, t)\}}{\max_{t \in S_v} \{d'_e(r, t) - d(r, t)\}}; \quad \rho_{\max} = \frac{\max_{t \in S_v} \{d_{e/e^*}(r, t)\}}{\max_{t \in S_v} \{d'_e(r, t)\}}.$$

Quite surprisingly, the best swap tree guarantees in the worst case good performances with respect to the above ratios, as reported in Table 1. Notice that such bounds are tight [12]. It is impressive to note that by using the **Swap-Increase** algorithm, the studied features of  $S_{e/e^*}(r)$  and  $S'_e(r)$  are very similar. Interestingly, this algorithm is also the cheapest in terms of time complexity. In the next section, we will see that this theoretical analysis is strengthened by the experience on real cases.

### 3 Experimental Results

In this section we present detailed experiments comparing the **Swap-Sum**, the **Swap-Increase** and the **Swap-Height** algorithm with the approach based on a dynamic recomputation proposed in [13], say **Dyn**, which is known to be efficient and easy to implement. Essentially, the **Dyn** algorithm is designed for the general

framework of dynamic graphs, where insertion and deletion of edges are allowed. In particular, when an edge  $e = (u, v)$  is deleted, the Dyn algorithm applies the general scheme of the Dijkstra's algorithm, by condensing the subgraph of  $G$  induced by subtree  $S_u$  to a single node. Hence, to compute  $S'_e(r)$  for every  $e \in E_S$ , we consider all the edges of the SPT one after the other: when the edge  $e$  of weight  $w(e)$  is taken into account, we increase its weight to an arbitrary large value, we compute the new SPT and we then set back the weight of  $e$  to  $w(e)$ . This will cost a total of  $O(nm + n^2 \log n)$  time and  $O(m)$  space, by implementing Dijkstra's algorithm using Fibonacci heaps [3].

The experiments were carried out in C++ programming language, with the support of the library of algorithms LEDA [10], and were performed on a PC with a Pentium® III 550 MHz, having 296 Mbytes of RAM and running the operating system Red Hat Linux® 6.2.

Let  $\delta(G) = \frac{2(m-n)}{n(n-3)}$ ,  $0 \leq \delta(G) \leq 1$ , denote the *edge density* of a 2-edge connected graph  $G$ , where  $\delta(G) = 0$  if  $G$  is a Hamiltonian cycle, and  $\delta(G) = 1$  if  $G$  is a clique. To test the swap algorithms, we have performed two different sets of experiments on random 2-edge connected weighted graphs. In the first set of experiments, we have considered graphs having 250 nodes and a variable density  $\delta(G) = i/10$ ,  $i = 1, \dots, 10$ , while in the second set of experiments, we have considered graphs of fixed density  $\delta(G) = 0.5$  and having a variable number of nodes  $n = 50 \cdot i$ ,  $i = 1, \dots, 10$ . Each random graph has been generated by starting from a cycle spanning all the nodes, and then adding a proper number of edges. The edge weights have been uniformly chosen on a nonnegative real interval, and the results presented have been averaged on 10 instances.

For each random graph, we have computed the SPT, and we have considered every edge on the SPT, in turn, as the faulty edge. This models a realistic scenario where each edge in the SPT can transiently fail with the same probability. To handle an edge failure, we have applied the three swap algorithms and the Dyn algorithm. For each swap algorithm the following values have been then measured:

- the weighted average over all the failing edges of  $\rho_\Sigma$ ,  $\rho_\Delta$  and  $\rho_{\max}$ , where the weight is the size of  $S_v$ ;
- the maximum value over all the failing edges of  $\rho_\Sigma$ ,  $\rho_\Delta$  and  $\rho_{\max}$ ;
- the used CPU time (in seconds) to compute all the best swap edges.

We have computed a weighted average of the ratios to be as much fair as possible, since the ratios tend to increase as soon as the size of  $S_v$  increases. In this way, the obtained values can be used to predict the performances of a best swap tree, once that the size of the detached subtree is known. Concerning the Dyn algorithm, we have computed the used CPU time (in seconds) to compute all the new SPTs, and the weighted average fraction of nodes in  $S_v$  changing their entering edge in the real new SPT, say  $\|S_v\|$ , where the weight is the size of  $S_v$  (motivations for computing a weighted average are analogous to those discussed for the ratios). This latter value is important to compare the two used approaches, since it measures the complexity of changes happening when rebuilding a real SPT (e.g., set-up costs, routing information, etc.).

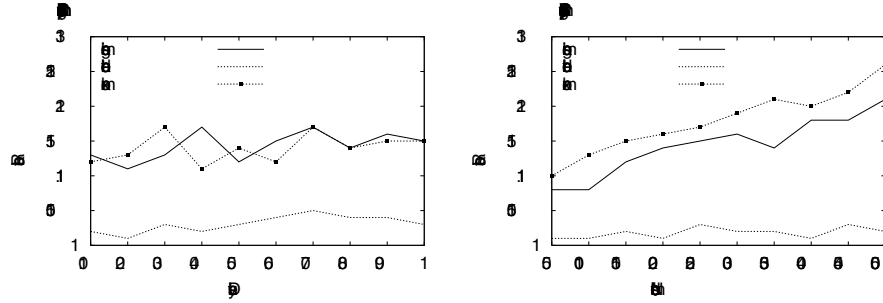


Fig. 2. Average ratios for the Swap-Sum algorithm.

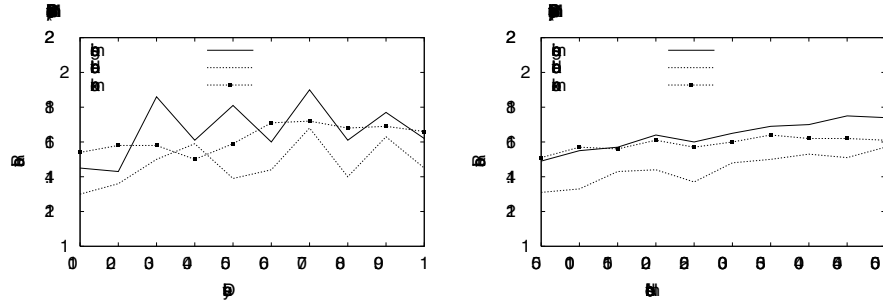


Fig. 3. Maximum ratios for the Swap-Sum algorithm.

For each figure presented in the following, the left (right) side refers to the first (second) set of experiments.

### 3.1 The Approximation Ratios

Figure 2-7 depict the linear interpolation of the results obtained for the weighted averages and the maximum values of  $\rho_\Sigma$  (rho-sigma),  $\rho_\Delta$  (rho-delta) and  $\rho_{\max}$  (rho-max).

More in detail, Figures 2 and 3 refers, respectively, to the average and maximum values obtained for the Swap-Sum algorithm. Looking to Figure 2, we observe that the average ratios are very close to 1, that is, a best swap tree computed to minimize  $F_{\{r,\Sigma\}}$  is functionally similar to the corresponding real new SPT. It is interesting to note that when the number of nodes is fixed and the density is variable (left side), then the approximation ratios increase very slowly as soon as the density increases. Actually, we have performed experiments for very sparse graphs as well, not reported here for space limitations, finding results very similar. Thus, the graph density seems to affect only to a limited extent the ratios. On the other hand, by looking to the right side of Figure 2, we observe that as soon as the number of nodes increase, the performances of the best swap tree decrease. Our interpretation is that for large graphs, when sub-

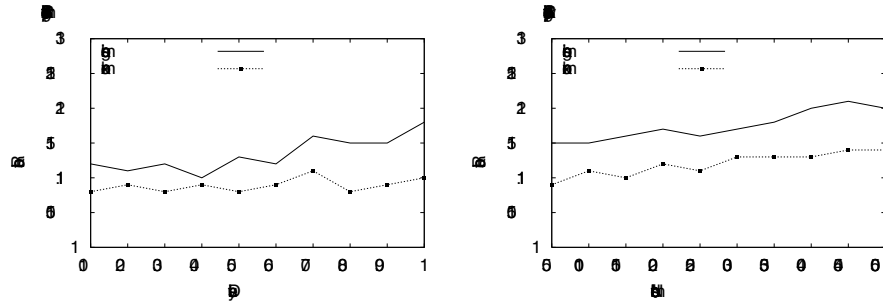


Fig. 4. Average ratios for the Swap-Increase algorithm.

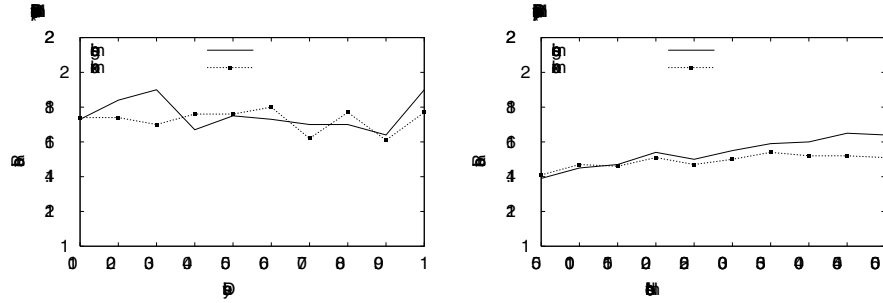


Fig. 5. Maximum ratios for the Swap-Increase algorithm.

trees containing many nodes are detached from the root, then the ratios become larger, thus affecting the weighted average. This behaviour has been observed by checking the ratios for large subtrees, which can be quite high (close to 2).

Focusing on Figure 3, we instead learn that the maximum ratios are maintained within 2, and thus also in the worst case, the best swap tree has a good functionality. Also in this case, the analysis for the left and right side of the figure is analogous.

Figures 4 and 5 refers, respectively, to the average and maximum values obtained for the **Swap-Increase** algorithm. Notice that in this case we have  $\rho_\Delta = 1$  by definition, and then values for  $\rho_\Delta$  are not reported. Also in this case, the average ratios are very close to 1, that is, a best swap tree computed to minimize  $F_{\{r, \Delta\}}$  is functionally similar to the corresponding real new SPT. Moreover, the maximum ratios are maintained within 2, as for the **Swap-Sum** algorithm. In both figures, the course of the ratios is analogous to that observed for the **Swap-Sum** algorithm. However, the ratios are always less than the corresponding ratios of the **Swap-Sum** algorithm, according to the theoretical analysis.

Finally, Figures 6 and 7 refers, respectively, to the average and maximum values obtained for the **Swap-Height** algorithm. In this case, results are very similar to those obtained for the **Swap-Sum** algorithm.

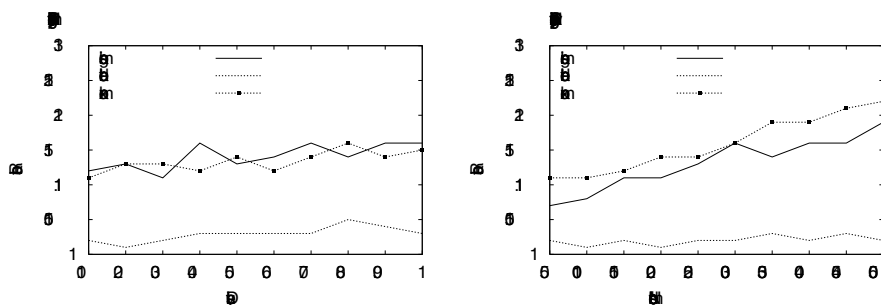


Fig. 6. Average ratios for the Swap-Height algorithm.

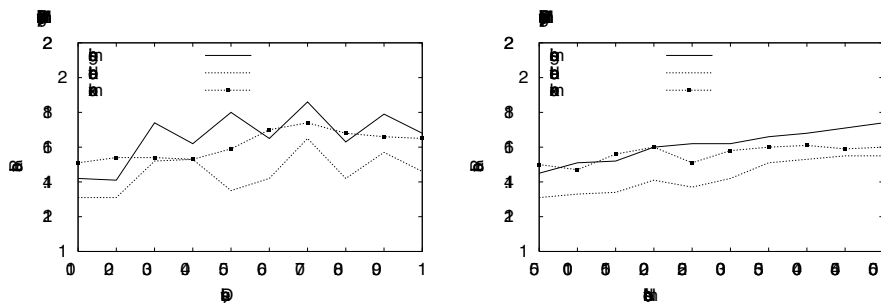


Fig. 7. Maximum ratios for the Swap-Height algorithm.

### 3.2 The CPU Time

Concerning the CPU time, the linear interpolation of the results is reported in Figure 8. Looking to the right side of the figure, we note that the time performances of all the algorithms show a superlinear growth, according to the theoretical analysis (in fact, a density of 0.5 means a quadratic number of edges). It is interesting to note that the **Swap-Height** algorithm performs worse than the **Swap-Sum** one, although it should be better in terms of asymptotic behaviour. This phenomenon is probably due to the fact that the former makes use of very complex data structures. In general, we observe that the **Dyn** algorithm performs better than what it was expected from the asymptotic analysis, even though it is generally outperformed by the **Swap-Sum** and the **Swap-Increase** algorithm.

### 3.3 Difference between the Old and the New SPT

Finally, Figure 9 shows the linear interpolations of the weighted average of the fraction  $\|S_v\|$  of nodes in  $S_v$  changing their entering edge in the new SPT.

It is worth noting that  $\|S_v\|$  is always quite close to 0.5, in both sets of experiments. In this case, by looking to the left side of the figure, we observe that  $\|S_v\|$  is more sensitive to the density of the graph, as it was expected. On



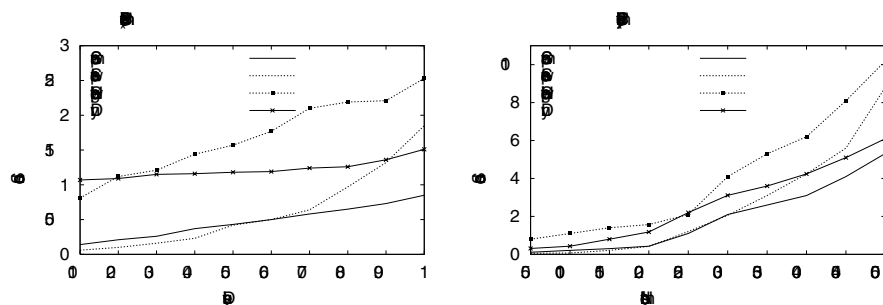
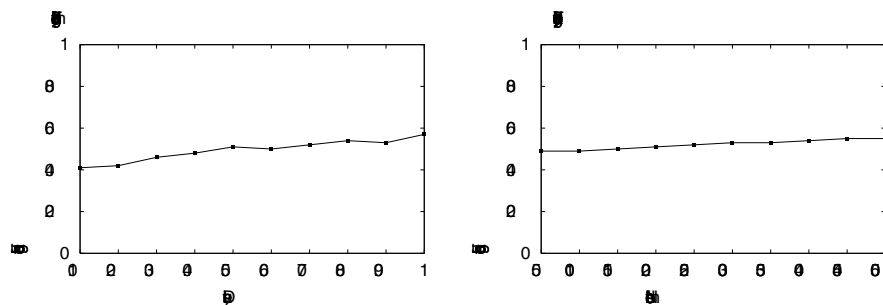


Fig. 8. CPU time for the swap algorithms as opposed to the Dyn algorithm.

Fig. 9. Fraction of nodes in  $S_v$  changing their entering edge in the new SPT.

the contrary, by looking to the right side, we now note that  $\|S_v\|$  grows slower, even though once again the number of nodes in  $G$  affects the weighted average.

From these results, we learn that the complexity of changes happening when rebuilding a real SPT is high. In fact, since we expect that half of the nodes in  $S_v$  change their parent in  $S'_e(r)$ , then we conclude that routing information are completely altered. Moreover, the set-up costs of  $S'_e(r)$  may be prohibitive, in case the size of  $S_v$  is large.

## 4 Conclusions

In this paper we have proposed an extensive experimental analysis of the so-called swap algorithms for a SPT. Our experiments showed that swapping is good and fast at the same time: good, since a best swap tree is functionally similar to the corresponding real new SPT, and fast, since the CPU time needed for computing all the best swap trees is generally less than that requested for recomputing dynamically all the new real SPTs. Moreover, the main result is that, while a best swap tree differs from the original SPT by just a single edge (i.e., the swap edge), a real new SPT requires in general the set-up of a number of new edges which is a substantial fraction (actually, about a half) of the number of nodes detached from the root as a consequence of the edge failure. We like

to emphasize that, in case of transient edge failures, this is for sure the first motivation in favour of swapping, rather than rebuilding a real new SPT.

In the near future, encouraged by these extremely positive results, we plan to extend our experiments to swap algorithms developed for other tree topologies. In particular, if the tree is a minimum diameter spanning tree (MDST), then it has been proved that a best swap MDST has a diameter at most  $5/2$  time longer than the diameter of a real new MDST [11], and we plan to verify whether this theoretical value is lower in the practice. Moreover, we will focus on the related problem of managing node failures in a SPT.

*Acknowledgements* – The author would like to thank Dino Di Paolo for his help in carrying out the experiments presented in the paper.

## References

1. F. Chin and D. Houck, Algorithms for updating minimal spanning trees, *J. Comput. System Sciences*, **16**(3) (1978) 333–344.
2. B. Dixon, M. Rauch and R.E. Tarjan, Verification and sensitivity analysis of minimum spanning trees in linear time, *SIAM J. Comput.*, **21**(6) (1992) 1184–1192.
3. M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. of the ACM*, **34**(3) (1987) 596–615.
4. D. Frigioni, A. Marchetti-Spaccamela and U. Nanni, Fully dynamic output bounded single source shortest path problem, *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, 1996, 212–221.
5. D. Frigioni, M. Ioffreda, U. Nanni and G. Pasquale, Experimental analysis of dynamic algorithms for the single-source shortest-path problem, *ACM J. of Experimental Algorithms*, **3**, article 5, (1998).
6. M. Grötschel, C.L. Monma and M. Stoer, Design of survivable networks, *Handbooks in OR and MS*, Vol. 7, Elsevier (1995) 617–672.
7. F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.
8. R. Hassin and A. Tamir, On the minimum diameter spanning tree problem, *Inf. Proc. Letters*, **53** (1995) 109–111.
9. G.F. Italiano and R. Ramaswami, Maintaining spanning trees of small diameter, *Algorithmica* **22**(3) (1998) 275–304.
10. K. Mehlhorn and S. Näher. *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, Cambridge, UK, 1999.
11. E. Nardelli, G. Proietti and P. Widmayer, Finding all the best swaps of a minimum diameter spanning tree under transient edge failures, *6th European Symp. on Algorithms (ESA'98)*, Springer-Verlag, LNCS 1461, 55–66, 1998.
12. E. Nardelli, G. Proietti and P. Widmayer, How to swap a failing edge of a single source shortest paths tree, *5th Annual Int. Computing and Combinatorics Conf. (COCOON'99)*, Springer-Verlag, LNCS 1627, 144–153, 1999.
13. G. Ramalingam and T. Reps, An incremental algorithm for a generalization of the shortest path problem, *J. of Algorithms*, **21** (1996) 267–305.
14. R.E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. of the ACM*, **22**(2) (1975) 215–225.
15. R.E. Tarjan, Sensitivity analysis of minimum spanning trees and shortest path trees, *Inf. Proc. Letters*, **14**(1) (1982) 30–33.

# Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study<sup>\*</sup>

Camil Demetrescu, Daniele Frigioni, Alberto Marchetti-Spaccamela, and  
Umberto Nanni

Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”,  
via Salaria 113, I-00198 Roma, Italy  
{demetres,frigioni,alberto,nanni}@dis.uniroma1.it

**Abstract.** We present the first experimental study of the fully dynamic single-source shortest paths problem in digraphs with arbitrary (negative and non-negative) arc weights. We implemented and tested several variants of the theoretically fastest fully dynamic algorithms proposed in the literature, plus a new algorithm devised to be as simple as possible while matching the best worst-case bounds for the problem. According to experiments performed on randomly generated test sets, all the considered dynamic algorithms are faster by several orders of magnitude than recomputing from scratch with the best static algorithm. The experiments also reveal that, although the simple dynamic algorithm we suggest is usually the fastest in practice, other dynamic algorithms proposed in the literature yield better results for specific kinds of test sets.

## 1 Introduction

The problem of finding efficient dynamic solutions for shortest paths has attracted a lot of interest in the last years, motivated by theoretical as well as practical applications. The problem is the following: we are given a graph  $G$  and we want to answer queries on the shortest paths of  $G$ , while the graph is changing due to insertions, deletions and weight updates of arcs. The goal is to update the information on the shortest paths more efficiently than recomputing everything from scratch after each update. If all the arc operations above are allowed, then we refer to the *fully dynamic* problem; if only insertions and weight decreases (deletions and weight increases) of arcs are supported, then we refer to the *partially dynamic incremental (decremental)* problem. The stated problem is interesting on its own and finds many important applications, including network optimization, document formatting, routing in communication systems, robotics. For a comprehensive review of the application settings for the static and dynamic shortest paths problem, we refer to [1] and [15], respectively.

Several theoretical results have been provided in the literature for the dynamic maintenance of shortest paths in graphs with positive arc weights (see,

---

<sup>\*</sup> Partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT).

e.g., [7,9,15,16]). We are aware of few efficient fully dynamic solutions for updating shortest paths in general digraphs with arbitrary (positive and non-positive) arc weights [10,16].

Recently, an equally important research effort has been done in the field of *algorithm engineering*, aiming at bridging the gap between theoretical results on algorithms and their implementation and practical evaluation. Many papers have been proposed in this field concerning the practical performances of *static* algorithms for shortest paths (see e.g. [4,5,13]), but very little is known for the experimental evaluation of *dynamic* shortest paths algorithms: [8] considers the fully dynamic single source shortest paths problem in digraphs with positive real arc weights. We are not aware of any experimental study in the case of arbitrary arc weights. On the other hand, several papers report on experimental works concerning different dynamic graph problems (see e.g., [2,3,11]).

In this paper we make a step toward this direction and we present the first experimental study of the fully dynamic single-source shortest paths problem in digraphs with arbitrary (negative and non-negative) arc weights. We implemented and experimented several algorithms for updating shortest paths in digraphs with arbitrary arc weights that undergo sequences of weight-increase and weight-decrease operations. Our main goal was that of identifying with experimental evidence the more convenient algorithm to use in practice in a fully dynamic setting. The starting points of our experimental study were the classical Bellman-Ford-Moore's algorithm (e.g., see [1]) and the fully dynamic algorithms proposed by Ramalingam and Reps in [15,16] and by Frigioni *et al.* in [10].

The solution in [15,16] requires that all the cycles in the digraph before and after any input update have positive length. It runs in  $O(|\delta| + |\delta| \log |\delta|)$  per update, where  $|\delta|$  is the number of nodes affected by the input change  $\delta$ , and  $||\delta||$  is the number of affected nodes plus the number of arcs having at least one affected endpoint. This gives  $O(m + n \log n)$  time in the worst case.

The algorithm in [10] has a worst case complexity per update that depends on the *output complexity* of the update operation and on a structural parameter of the graph called *k-ownership*. Weight-decrease operations require  $O(\min\{m, kn_a\} \log n)$  worst case time, while weight-increase operations require  $O(\min\{m \log n, k(n_a + n_b) \log n + n\})$  worst case time. Here  $n_a$  is the number of affected nodes, and  $n_b$  is the number of nodes considered by the algorithm and maintaining both the distance and the parent in the shortest paths tree.

The common idea behind these algorithms is to use a technique of Edmonds and Karp [6], which allows it to transform the weight of each arc in a digraph into a non-negative real without changing the shortest paths, and to apply an adaptation of Dijkstra's algorithm to the modified graph. Differently from the case where all arc weights are non-negative (for which no efficient dynamic worst-case solution is known), with this technique it is possible to reduce from  $O(mn)$  to  $O(m + n \log n)$  the worst-case time of updating a shortest paths tree after a change of the weight of an arc in a graph with  $n$  nodes and  $m$  arcs.

As a first contribution of the paper, we confirm this claim from an experimental point of view. In particular, we observed that on randomly generated test sets, dynamic algorithms based on the technique of Edmonds and Karp

are experimentally faster by several orders of magnitude than recomputing from scratch using the best static algorithm.

The paper also suggests a simple dynamic algorithm that hinges upon the technique of Edmonds and Karp without using complex data structures. The algorithm was devised to be as simple as possible while matching the  $O(m + n \log n)$  bound of the best previous dynamic algorithms for the problem.

We implemented and experimentally evaluated all the aforementioned algorithms with the goal of improving their performance in practice. Experiments performed on randomly generated test sets showed that, though our simple dynamic algorithm is usually the fastest in practice, both the algorithms of Ramalingam and Reps and a simplified version of the algorithm of Frigioni *et al.* yield better results for specific kinds of test sets, e.g., where the range of values of arc weights is small. Our implementations were written in C++ with the support of LEDA [14]. The experimental platform including codes, test sets generators and results can be accessed over the Internet at the URL: <ftp://www.dis.uniroma1.it/pub/demetres/experim/dsplib-1.1/> and was designed to make experiments easily repeatable.

## 2 Algorithms under Evaluation

Let  $G = (N, A, w)$  be a weighted directed graph with  $n = |N|$  nodes and  $m = |A|$  arcs, where  $w$  is function that associates to each  $(x, y) \in A$  a real weight  $w_{x,y}$ , and let  $s \in N$  be a fixed *source* node. If  $G$  does not contain negative cycles, then, for each  $x \in N$ , we denote as  $d(x)$  the minimum distance of  $x$  from  $s$ , and as  $T(s)$  a shortest paths tree of  $G$  rooted at  $s$ . For each  $x \in N$ ,  $T(x)$  denotes the subtree of  $T(s)$  rooted at  $x$ ,  $p(x)$  denotes the parent of  $x$  in  $T(s)$ , and  $\text{IN}(x)$  and  $\text{OUT}(x)$  denote the arcs of  $A$  incoming and outgoing  $x$ , respectively. The well known optimality condition of the distances of the nodes of a digraph  $G = (N, A)$  states that, for each  $(z, q) \in A$ ,  $d(q) \leq d(z) + w_{z,q}$  (see, e.g., [1]). The new shortest paths tree in the graph  $G'$ , obtained from  $G$  after an arc update, is denoted as  $T'(s)$ , while  $d'(x)$  and  $p'(x)$  denote the distance and the parent of  $x$  after the update, respectively.

We assume that the digraph  $G$  before an arc update does not contain negative cycles, and consider digraphs that undergo sequences of *decrease* and *increase* operations on the weights of arcs (*insert* and *delete* operations, respectively, can be handled analogously). We say that a node is *affected* by an input update if it changes the distance from the source due to that update.

Every time a dynamic change occurs in the digraph, we have two possibilities to update the shortest paths: either we recompute everything from scratch by using the best static algorithm, or we apply dynamic algorithms. In the following we analyze in detail these possibilities.

### 2.1 Static Algorithms

The best static algorithm for solving the shortest paths problem in the case of general arc weights is the classical Bellman-Ford-Moore's algorithm [1,14] (in short, **BFM**). Many different versions of **BFM** have been provided in the literature

(see [1] for a wide variety). The worst case complexity of all these variants is  $O(mn)$ . In [5] the authors show that the practical performances of BFM can be improved by using simple heuristics. In particular, they show that the heuristic improvement of BFM given in [13] is the fastest in practice. However, from a theoretical point of view, nothing better than the  $O(mn)$  worst case bound is known. In our experiments, we considered the LEDA implementation of BFM.

## 2.2 Fully Dynamic Algorithms

We implemented the following fully dynamic algorithms: 1) the algorithm in [10], referred as FMN; 2) the algorithm in [16], referred as RR; 3) a simple variant of FMN, denoted as DFMN; 4) a new simple algorithm we suggest, denoted as DF.

The common idea behind all these algorithms is to use a technique of Edmonds and Karp [6], which allows it to transform the weight of each arc in a digraph into a non-negative real without changing the shortest paths. This is done as follows: after an input update, for each  $(z, v) \in A$ , replace  $w_{z,v}$  with the *reduced weight*  $r_{z,v} = d(z) + w_{z,v} - d(v)$ , and apply an adaptation of Dijkstra's algorithm to the modified graph. The computed distances represent changes to the distances since the update. The actual distances of nodes after the update can be easily recovered from the reduced weights. This allows it to reduce from  $O(mn)$  to  $O(m + n \log n)$  the worst-case time of updating a shortest paths tree after a change of the weight of an arc in a digraph with  $n$  nodes and  $m$  arcs.

In what follows we give the main idea of the implemented algorithms to handle *decrease* and *increase* operations. For more details we refer to [10,15,16].

*Weight decrease operations.* Concerning the case of a *decrease* operation on arc  $(x, y)$ , all the implemented algorithms basically update the shortest paths information by a Dijkstra's computation performed starting from node  $y$ , according to the technique of Edmonds and Karp. In Dijkstra's computation, when a node  $z$  is permanently labeled, *all* arcs  $(z, h)$  are traversed and the priority of  $h$  in the priority queue is possibly updated.

The only exception concerns FMN, where the following technique is exploited to bound the number of traversed arcs. For each node  $z$ , the sets  $\text{IN}(z)$  and  $\text{OUT}(z)$  are partitioned into two subsets as follows. For each  $x \in N$ ,  $\text{IN-OWN}(x)$  denotes the subset of  $\text{IN}(x)$  containing the arcs owned by  $x$ , and  $\overline{\text{IN-OWN}}(x) = \text{IN}(x) - \text{IN-OWN}(x)$  denotes the set of arcs in  $\text{IN}(x)$  not owned by  $x$ . Analogously,  $\text{OUT-OWN}(x)$  and  $\overline{\text{OUT-OWN}}(x)$  represent the arcs in  $\text{OUT}(x)$  owned and not owned by  $x$ , respectively. Digraph  $G$  admits a  $k$ -ownership if, for all nodes  $x$ , both  $\text{IN-OWN}(x)$  and  $\text{OUT-OWN}(x)$  contain at most  $k$  arcs (see [9] for more details). Finally, the arcs in  $\overline{\text{IN-OWN}}(x)$  ( $\overline{\text{OUT-OWN}}(x)$ ) are stored in a min-based (max-based) priority queue where the priority of arc  $(y, x)$  ( $(x, y)$ ) is the quantity  $d(y) + w_{y,x}$  ( $d(y) - w_{x,y}$ ). When the new distance of a node  $z$  is computed the above partition allows it to traverse only the arcs  $(z, h)$  in  $\text{OUT-OWN}(z)$  and those in  $\overline{\text{OUT-OWN}}(z)$ , such that  $h$  is affected as well. This is possible by exploiting the priority of the arcs in  $\overline{\text{OUT-OWN}}(z)$ .

*Weight increase operations.* In the case of an *increase* of the weight of on arc  $(x, y)$  of a positive quantity  $\epsilon$ , the implemented algorithms work in two phases.

First they find the affected nodes and then compute the new distances for the affected nodes. The second phase is essentially the same for all the algorithms, and consists of a Dijkstra's computation on the subgraph of  $G$  induced by the affected nodes, according to the technique of Edmonds and Karp. The main differences concern the first phase. As we will see, the only exception concerns DF, which avoids computing the first phase.

- FMN. The first phase of FMN is performed by collecting the nodes in a set  $M$ , extracting them one by one, and searching an alternative shortest path from  $s$ . To this aim, for each affected node  $z$  considered, only the arcs  $(h, z)$  in  $\text{IN-OWN}(z)$  and those in  $\overline{\text{IN-OWN}}(z)$ , such that  $h$  is affected as well, are traversed. This is possible by exploiting the priority of the arcs in  $\overline{\text{IN-OWN}}(z)$ . This phase is quite complicated since it also handles zero cycles in an output bounded fashion.

- DFMN. The main difference of DFMN with respect to FMN is the elimination of the partition of arcs in owned and not-owned, that increases the number of arcs traversed (wrt FMN), but allows us to obtain a simpler and faster code.

- RR. Concerning RR, observe that it maintains a subset  $SP$  of the arcs of  $G$ , containing the arcs of  $G$  that belong to at least one shortest path from  $s$  to the other nodes of  $G$ . The digraph with node set  $N$  and arc set  $SP$  is a dag denoted as  $SP(G)$ . As a consequence, RR works only if all the cycles in the digraph, before and after any input update, have positive length. In fact, if zero cycles are allowed, then all of these cycles that are reachable from the source will belong to  $SP(G)$ , which will no longer be a dag. The first phase of RR finds the affected nodes as follows. It maintains a work set containing nodes that have been identified as affected, but have not yet been processed. Initially,  $y$  is inserted in that set only if there are no further arcs in  $SP(G)$  entering  $y$  after the operation. Nodes in the work set are processed one by one, and when node  $u$  is processed, all arcs  $(u, v)$  leaving  $u$  are deleted from  $SP(G)$ , and  $v$  is inserted in the work set. All nodes that are identified as affected during this phase are inserted in the work set.

- DF. Now we briefly describe the main features of DF, in the case of an *increase* operation. DF maintains a shortest paths tree of the digraph  $G$ , and is able to detect the introduction of a negative cycle in the subgraph of  $G$  reachable from the source, as a consequence of an insert or a *decrease* operation. Zero cycles do not create any problem to the algorithm. Differently from RR and FMN, the algorithm has not been devised to be efficient in output bounded sense, but to be fast in practice, and costs  $O(m+n \log n)$  in the worst case. The algorithm consists of two phases called **Initializing** and **Updating**. The **Initializing** phase marks the nodes in  $T(y)$  and, for each marked node  $v$ , finds the best unmarked neighbor  $p$  in  $\text{IN}(v)$ . This is done to find a path (not necessarily a shortest path) from  $s$  to  $v$  in  $G'$  whose length is used to compute the initial priority of  $v$  in the priority queue  $H$  of the **Updating** phase. If  $p \neq \text{nil}$  and  $d(p) + w_{p,v} - d(v) < \epsilon$  then this priority is computed as  $d(p) + w_{p,v} - d(v)$ , otherwise it is initialized to  $\epsilon$ , which is the variation of  $y$ 's distance. In both cases the initial priority of the node is an upper bound on the actual variation. The **Updating** phase properly updates the information on the shortest paths from  $s$  to the marked nodes by performing a computation analogous to Dijkstra's algorithm.

In general, FMN and RR perform the Dijkstra's computation on a set of nodes which is a subset of the nodes considered by the **Updating** phase of DF. The **Initializing** phase of DF simply performs a visit of the subgraph of  $G$  induced by the arcs in  $\text{IN}(z)$ , for each node  $z$  in  $T(y)$ , in order to find a temporary parent of  $z$  in the current shortest paths tree. This shows that DF is not output bounded.

### 3 Experimental Setup

In this section we describe our experimental framework, presenting the problem instances generators, the performance indicators we consider, and some relevant implementation details. All codes being compared have been implemented by the authors as C++ classes using advanced data types of LEDA [14] (version 3.6.1). Our experiments were performed on a SUN Workstation Sparc Ultra 10 with a single 300 MHz processor and 128 MB of main memory running UNIX Solaris 5.7. All C++ programs were compiled by the GNU g++ compiler version 1.1.2 with optimization level O4. Each experiment consisted of maintaining both the distance of nodes from the source and the shortest paths tree in a random directed graph by means of algorithms BFM, FMN, DFMN, RR and DF upon a random mixed sequence of *increase* and *decrease* operations. In the special case of BFM, after each update the output structures were rebuilt from scratch.

#### 3.1 Graph and Sequence Generators

We used four random generators for synthesizing the graphs and the sequences of updates:

- **gen\_graph**( $n, m, s, \min, \max$ ): builds a random directed graph with  $n$  nodes,  $m$  arcs and integer arc weights  $w$  s.t.  $\min \leq w \leq \max$ , forming no negative or zero length cycle and with all nodes reachable from the source node  $s$ . Reachability from the source is obtained by first generating a connecting path through the nodes as suggested in [5]; remaining arcs are then added by uniformly and independently selecting pairs of nodes in the graph. To avoid introducing negative and zero length cycles we use the potential method described in [12].
- **gen\_graph\_z**( $n, m, s, \min, \max$ ): similar to **gen\_graph**, but all cycles in the generated graphs have exactly length zero.
- **gen\_seq**( $G, q, \min, \max$ ): issues a mixed sequence of  $q$  *increase* and *decrease* operations on arcs chosen at random in the graph  $G$  without introducing negative and zero length cycles. Weights of arcs are updated so that they always fit in the range  $[\min, \max]$ . Negative and zero length cycles are avoided by using the same potentials used in the generation of weights of arcs of  $G$ . Optionally, the following additional constraints are supported:
  - *Modifying Sequence*: each increase or decrease operation is chosen among the operations that actually modify some shortest path from the source.
  - *Alternated Sequence*: the sequence has the form *increase-decrease-increase-decrease...*, where each pair of consecutive *increase-decrease* updates is performed on the same arc.



- `gen_seq_z(G, q, min, max)`: similar to `gen_seq`, but the update operations in the generated sequences force cycles in the graph  $G$  to have length zero.

All our generators are based on the LEDA pseudo-random source of numbers. We initialized the random source with a different odd seed for each graph and sequence we generated.

### 3.2 Performance Indicators

We considered several performance indicators for evaluating and comparing the different codes. In particular, for each experiment and for each code we measured: (a) the average running time per update operation during the whole sequence of updates; (b) the average number of nodes processed in the distance-update phase of algorithms. Again, this is per update operation during the whole sequence of updates.

Indicator (b) is very important in an output-bounded sense as it measures the actual portion of the shortest paths tree for which the dynamic algorithms perform high-cost operations such as extractions of minima from a priority queue. It is interesting to observe that, if an *increase* operation is performed on an arc  $(x, y)$ , the value of the indicator (b) measured for both **RR** and **DFMN** reports the number of affected nodes that change their distance from the source after the update, while the value of (b) measured for **DF** reports the number of nodes in the shortest paths tree rooted at  $y$  before the update.

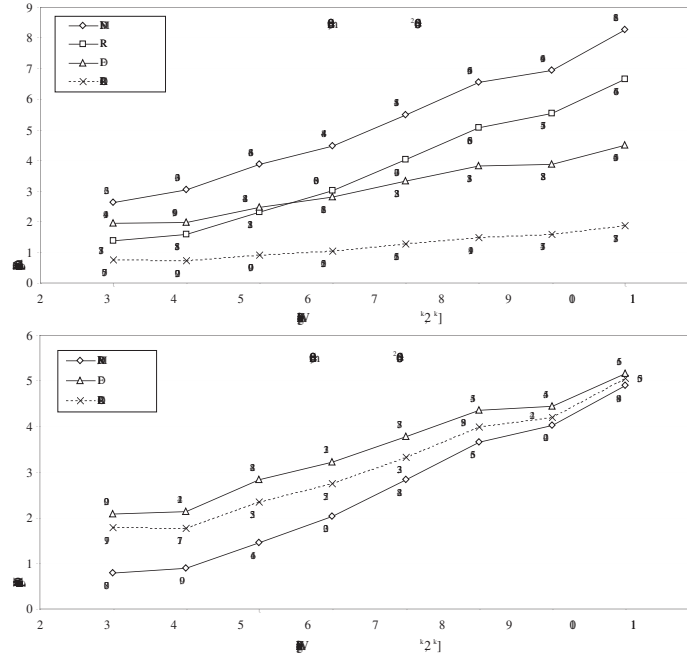
Other measured indicators were: (c) the maximum running time per update operation; (d) the average number of scanned arcs per update operation during the whole sequence of updates; (e) the total time required for initializing the data structures.

The running times were measured by the UNIX system call `getrusage()` and are reported in milliseconds. Indicators (b) and (d) were measured by annotating the codes with probes. The values of all the indicators are obtained by averaging over 15 trials. Each trial consists of a graph and a sequence randomly generated through `gen_graph` or `gen_graph_z` and `gen_seq` or `gen_seq_z`, respectively, and is obtained by initializing the pseudo-random generator with a different seed.

### 3.3 Implementation Details

We put effort to implementing algorithms **DFMN**, **RR** and **DF** in such a way that their running times can be compared as fairly as possible. In particular, we avoided creating “out of the paper” implementations of algorithms **DFMN** and **RR**. For example, in **RR** we do not explicitly maintain the shortest paths dag  $SP$  so as to avoid additional maintenance overhead that may penalize **RR** when compared with the other algorithms. Instead, we tried to keep in mind the high-level algorithmic ideas while devising fast codes.

For these reasons, we used just one code for performing *decrease* and we focused on hacking and tweaking codes for *increase*. We believe that the effect of using LEDA data structures does not affect the relative performance of different algorithms. More details about our codes can directly be found in our experimental package distributed over the Internet. In the remainder of this paper, we



**Fig. 1.** Experiments performed with  $n = 300$  and  $m = 0.5n^2 = 45000$  for arc weight intervals increasing from  $[-8, 8]$  to  $[-1024, 1024]$ .

refer to ALL-DECR as the *decrease* code and to DFMN, RR and DF as the *increase* codes.

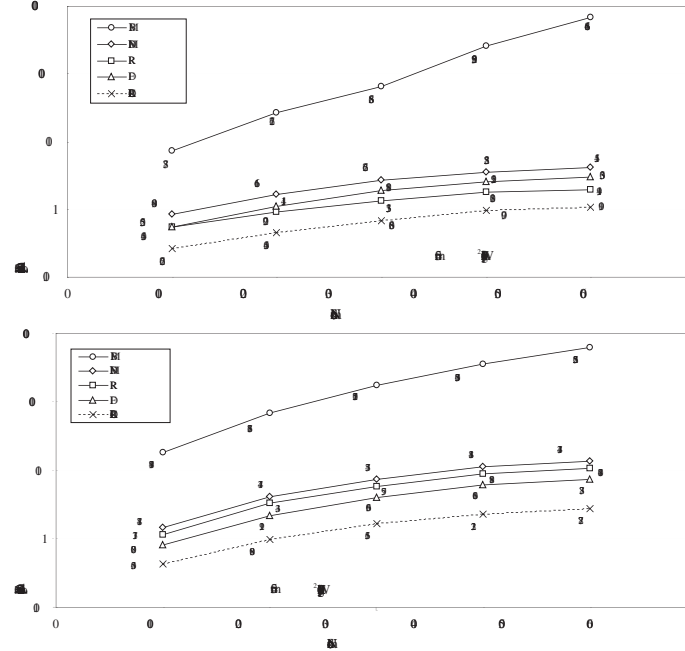
## 4 Experimental Results

The goal of this section is to identify with experimental evidence the more convenient algorithm to use in practice in a fully dynamic environment. We performed several experiments on random graphs and random update sequences for different parameters defining the test sets aiming at comparing and separating the performances of the different algorithms.

Preliminary tests proved that, due to its complex data structures, FMN is not practical neither for *decrease* nor for *increase* operations, and so we focused on studying the performances of its simplified version DFMN.

Our first experiment showed that the time required by an *increase* may significantly depend upon the width of the interval of the arc weights in the graph:

- *Increasing arc weight interval:* we ran our DFMN, RR and DF codes on mixed sequences of 2000 modifying update operations performed on graphs with 300 nodes and  $m = 0.5n^2 = 45000$  arcs and with arc weights in the range  $[-2^k, 2^k]$  for values of  $k$  increasing from 3 to 10. The results of this test for *increase* operations are shown in Figure 1. It is interesting to note that the smaller is the width of the arc weight interval, the larger is the gap

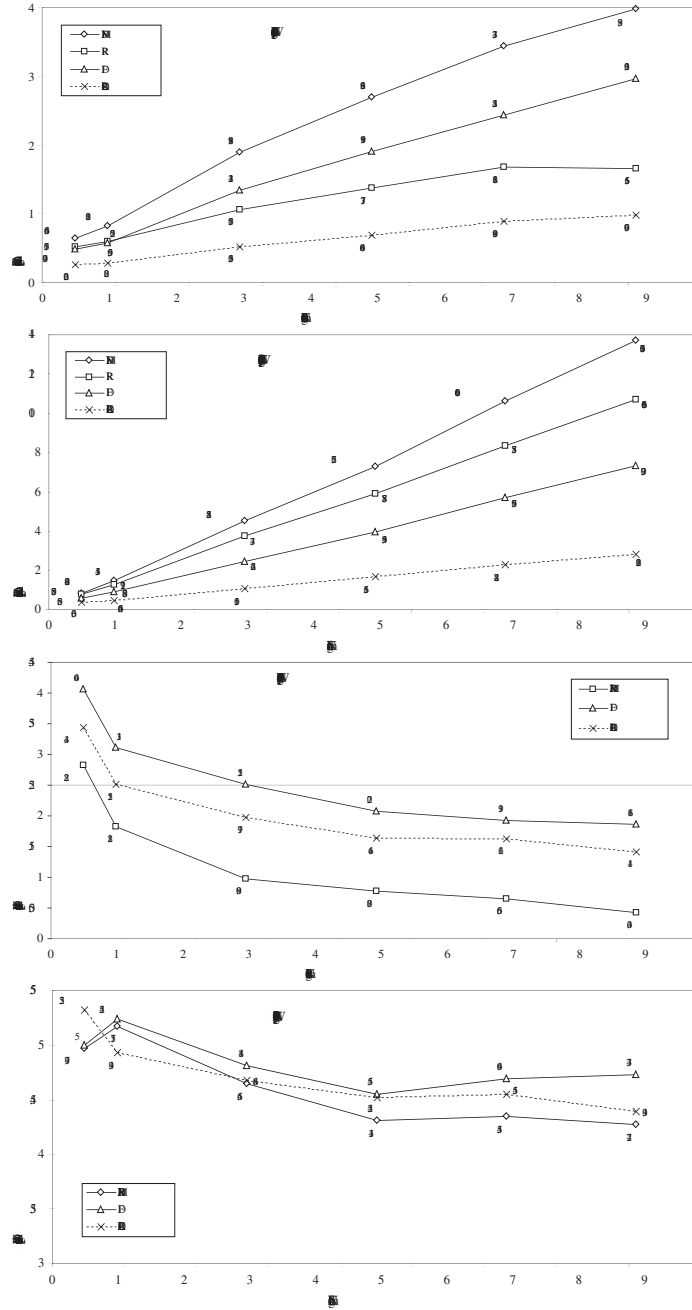


**Fig. 2.** Experiments performed with  $100 \leq n \leq 500$  and  $m = 0.5n^2$  for arc weights in the range  $[-10, 10]$  and  $[-1000, 1000]$ .

between the number of affected nodes considered by RR during any *increase* operation on an arc  $(x, y)$ , and the number of nodes in  $T(y)$  scanned by DF. In particular, RR is faster than DF for weight intervals up to  $[-32, 32]$ , while DF improves upon RR for larger intervals. This experimental result agrees with the fact that RR is theoretically efficient in output bounded sense, but spends more time than DF for identifying affected nodes. The capacity of identifying affected nodes even in presence of zero cycles penalizes DFMN that is always slower than RR and DF on these problem instances with no zero cycles.

In our second suite of experiments, we ran BFM, DFMN, RR and DF codes on random sequences of 2000 modifying updates performed both on dense and sparse graphs with no negative and zero cycles and for two different ranges of the arcs weights. In particular, we performed two suites of tests:

- *Increasing number of nodes:* we measured the running times on dense graphs with  $100 \leq n \leq 500$  and  $m = 0.5n^2$  for arc weights in  $[-10, 10]$  and  $[-1000, 1000]$ . We repeated the experiment on larger sparse graphs with  $1000 \leq n \leq 3000$  and  $m = 30n$  for the same arc weights intervals and we found that the performance indicators follow the same trend of those of dense graphs that are shown in Figure 2. This experiment agrees with the first one and confirms that, on graphs with arc density 50%, DF beats RR for large weight intervals and RR beats DF for small weight intervals. Notice that the



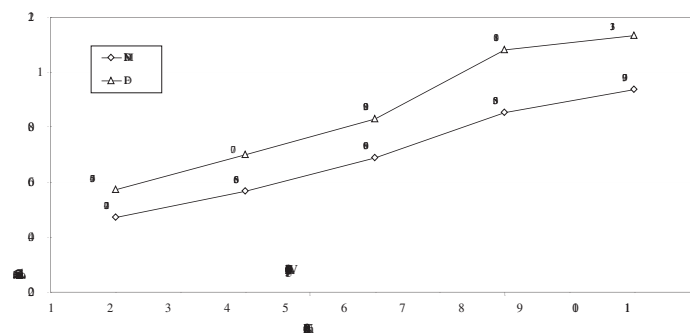
**Fig. 3.** Experiments performed with  $n = 300$ ,  $0.05n^2 \leq m \leq 0.9n^2$  for arc weights in the range  $[-10, 10]$  and  $[-1000, 1000]$ .

dynamic codes we considered are better by several orders of magnitude than recomputing from scratch through the LEDA BFM code.

- *Increasing number of arcs*: we retained both the running times and the number of nodes processed in the distance-update phase of algorithms on dense graphs with  $n = 300$  and  $0.05n^2 \leq m \leq 0.9n^2$  for arc weights in  $[-10, 10]$  and  $[-1000, 1000]$ . We repeated the experiment on larger sparse graphs with  $n = 2000$  and  $10n \leq m \leq 50n$  for the same arc weights intervals and again we found similar results. Performance indicators for this experiment on dense graphs are shown in Figure 3 and agree with the ones measured in the first test for what concerns the arc weight interval width. However, it is interesting to note that even for small weight ranges, if the arc density is less than 10%, the running time of DF slips beneath that of RR.

As from the previous tests our DFMN code is always slower than RR and DF, our third experiment aims at investigating if families of problem instances exist for which DFMN is a good choice for a practical dynamic algorithm. As it is able to identify affected nodes even in presence of zero cycles, we were not surprised to see that DFMN beats in practice DF in a dynamic setting where graphs have many zero cycles. We remark that RR is not applicable in this context.

- *Increasing number of arcs and zero cycles*: we ran DFMN and DF codes on random graphs with 2000 nodes,  $10n \leq m \leq 50n$ , weights in  $[-10, 10]$ , all zero cycles, and subject to 2000 random alternated and modifying updates per sequence. We used generators `gen_graph.z` and `gen_seq.z` to build the input samples. Figure 4 shows the measured running times of increase operations for this experiment.



**Fig. 4.** Experiments performed with  $n = 2000$ ,  $10n \leq m \leq 50n$  for arc weights in  $[-10, 10]$ . All cycles have zero length during updates.

Performance indicators (c), (d) and (e) provided no interesting additional hint on the behavior of the algorithms and therefore we omit them from our discussion: the interested reader can find in the experimental package the detailed results tables of our tests.

## 5 Future Work

The continuation of the present work is along two directions: (1) Performing experiments on graphs from real life; (2) Reimplementing the algorithms in the C language, without the support of LEDA, with the goal of testing algorithms on larger data sets.

## References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.
2. D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. *ACM Journal on Experimental Algorithmics*, 2:Article 5, 1997.
3. G. Amato, G. Cattaneo, and G. F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In *ACM-SIAM Symp. on Discrete Algorithms*, pp. 1–10, 1997.
4. B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. In *European Symp. on Algorithms*. Lect. Notes in Comp. Sc. 1136, pp. 349–363, 1996.
5. B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.
6. J. Edmonds, R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.
7. P. G. Franciosa, D. Frigioni, and R. Giaccio. Semi-dynamic shortest paths and breadth-first search on digraphs. In *Symp. on Theoretical Aspects of Computer Science*. Lect. Notes in Comp. Sc. 1200, pp. 33–46, 1997.
8. D. Frigioni, M. Ioffreda, U. Nanni, G. Pasqualone. Experimental Analysis of Dynamic Algorithms for the Single Source Shortest Path Problem. *ACM Journal on Experimental Algorithmics*, 3:Article 5 (1998).
9. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):351–381, 2000.
10. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *European Symp. on Algorithms*. Lect. Notes in Comp. Sc. 1461, pp. 320–331, 1998.
11. D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Shaefer, C. Zaroliagis. An experimental study of dynamic algorithms for directed graphs. In *European Symp. on Algorithms*. Lect. Notes in Comp. Sc. 1461, pp. 368–380, 1998.
12. A. V. Goldberg. Selecting problems for algorithm evaluation. In *Workshop on Algorithm Engineering*. Lect. Notes in Comp. Sc. 1668, pp. 1–11, 1999.
13. A. V. Goldberg, and T. Radzik. A heuristic improvement of the Bellman-Ford algorithm. *Applied Math. Letters*, 6:3–6, 1993.
14. K. Mehlhorn and S. Naher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.
15. G. Ramalingam. Bounded incremental computation. Lect. Notes in Comp. Sc. 1089, 1996.
16. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

# New Algorithms for Examination Timetabling

Massimiliano Caramia<sup>1,\*</sup>, Paolo Dell’Olmo<sup>2,\*\*</sup>, and Giuseppe F. Italiano<sup>1,\*\*\*</sup>

<sup>1</sup> Dipartimento di Informatica, Sistemi e Produzione,  
Università di Roma “Tor Vergata”, Roma, Italy  
`{caramia,italiano}@disp.uniroma2.it`

<sup>2</sup> Dipartimento di Statistica, Università di Roma “La Sapienza”, Roma, Italy  
`dellolmo@pow2.sta.uniroma1.it`

**Abstract.** In examination timetabling a given set of examinations must be assigned to as few time slots as possible so as to satisfy certain side constraints and so as to reduce penalties deriving from proximity constraints. In this paper, we present new algorithms for this problem and report the results of an extensive experimental study. All our algorithms are based on local search and are compared with other existing implementations in the literature.

## 1 Introduction

Examination timetabling is an important operations management problem in Universities worldwide and has been extensively investigated (see e.g., [1]–[9], [11]–[15]). Although educational systems are quite different in different countries, the task of scheduling examinations for University courses shares a common framework: the basic problem is to assign examinations to a certain number of time slots in such a way that there are no conflicts (e.g., no student has to take more than one examination at a time). The complexity of this problem depends on the degrees of freedom that students are allowed in their choice of courses: a greater freedom of choices typically increases the difficulty of producing an examination schedule that fits into a certain number of time slots without creating timetable conflicts for some students. There are also other issues, related to the quality of the solution, which must be taken into account: for instance, among two different timetables having the same number of time slots, one might prefer the one having examinations spaced out more evenly or the one with lower number of undesirable assignments. For these reasons, timetabling problems make use of certain cost measures to assess the quality of potential schedules. In most cases there is an additional *proximity cost*  $w_t$  whenever a student has to take two

---

\* Work partially supported by the Italian National Research Council (CNR).

\*\* Work partially supported by the Italian National Research Council (CNR) under contract n. 98.00770.CT26.

\*\*\* Work partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT), by the Italian Ministry of University and Scientific Research (Project “Algorithms for Large Data Sets: Science and Engineering”), and by the Italian National Research Council (CNR).

examinations scheduled  $t$  time slots apart. The proximity cost is non-decreasing with the time slot difference, i.e., the closer two conflicting examinations are scheduled, the higher is the penalty induced by their proximity cost.

In general, the objective of examination timetabling is twofold. First of all, one wishes to find a conflict-free solution which schedules all exams in as few time slots as possible. Secondly, an important measure of the quality of a solution is the ability to spread student examinations as evenly as possible over the schedule, i.e., with the minimum cost. These two goals are clearly in contrast: the fewer the time slots, the less the examinations can be spread out evenly, and the more the penalty introduced by proximity costs. We mention that there are also other *side constraints* which might play a role into examination timetabling. For instance, there can be limitations on available classroom capacity or specific classroom availabilities. It is easy to see that examination timetabling problem, even in its basic versions, is  $\mathcal{NP}$ -complete as it is a generalization of graph coloring [10].

**Related Work.** Among the extensive literature on examination timetabling (see e.g., [1]–[9], [11]–[15]), we only cite the most recent implementations, which are by Carter *et al.* [7], by Burke *et al.* [2], and by Di Gaspero and Schaerf [9]. Carter *et al.* [7] employed known graph coloring techniques to tackle examination timetabling without side constraints. They solved an underlying graph coloring problem, where examinations were sorted in order of decreasing difficulty according to some criterion: examinations were assigned to time slots without creating conflicts, in the order they were processed. To reduce the length of the schedule obtained, some form of backtracking was used. Five different algorithms, according to the sorting criterion chosen (and referred to as LD, SD, LWD, LE and RO) were implemented. They also performed extensive experiments where only proximity constraints (and their associated penalties) are taken into account. The details of the method are spelled out in [7]. Burke *et al.* [2] designed a *memetic* timetabling algorithm based on evolutionary techniques, which deals exclusively with fixed length timetables. Their algorithm is based on the correct use of a number of evolutionary operators on a population chosen by a selection operator. The experiments in [2] considered both proximity constraints and classroom capacity constraints: i.e., a maximum number of available seats per time slot is given, and thus only a limited number of students can be examined in each time slot. Very recently Di Gaspero and Schaerf [9] proposed a family of algorithms based on several variants of tabu search.

**Our Results.** Most of the timetabling algorithms proposed in the literature are designed either to minimize the number of time slots (see e.g., [7]) or to minimize the overall penalty using a fixed number of time slots (see e.g., [2]). In this paper, we consider both issues and investigate in more depth the tradeoffs between time slots and penalties: we propose a family of timetabling algorithms based on local search which have quite different behavior with respect to time slots and penalties. We perform an extensive experimental study of our algorithms, and compare them to previous approaches. We try to reproduce the same experimental scenario set up by previous work: namely, we consider proximity constraints,



room capacity constraints, and test our implementations on several real-world benchmarks for timetabling examination available in the public domain.

## 2 Our Timetabling Algorithms

Given a set of examinations  $E$ , *conflict graph*  $G$  can be defined as follows. There is a vertex in the conflict graph for each examination, and there is an edge between vertex  $i$  and vertex  $j$  if at least one student is enrolled in both exams  $i$  and  $j$ . In what follows, we will use interchangeably the term vertex and exam, as there is no danger of ambiguity. All our algorithms for examination timetabling perform local search on this conflict graph, driven by the following variables associated with each exam  $i$ .

- $t_i$  : The time slot  $t_i$  tentatively assigned to exam  $i$ .
- $p_i$  : The priority with which exam  $i$  is examined. At the beginning,  $p_i$  is initialized with the degree of vertex  $i$  in the conflict graph, i.e., exams with higher potential of conflicts are tentatively scheduled first.
- $z_i$  : A penalty given by the sum of proximity costs scored by  $i$  in the current timetable. At the beginning,  $z_i$  is initialized with the sum of proximity costs given by the case where exam  $i$  is scheduled with the minimum distance from the other incompatible exams.

Our algorithm starts with a *greedy\_scheduler*, which considers exams by their non-increasing priorities: exam  $i$  is allocated in the lowest available time slot it can be assigned to without generating conflicts. At the end, each penalty  $z_i$  is updated accordingly to the schedule found. Let  $T = \max_{i \in E} \{t_i\}$  be the total number of time slots used in the current timetable. After this step, a *penalty\_decreaser* tries to improve the quality of the solution without increasing  $T$ . To accomplish this task, *penalty\_decreaser* attempts to space out more evenly the exams while keeping the same number of time slots. In particular, exams are considered by non-increasing penalty: exam  $i$  is scheduled to a different time slot (not exceeding  $T$ ) when this new assignment decreases the overall penalty of the timetable. If *penalty\_decreaser* is not successful, i.e., it is not able to change the schedule, it calls a *penalty\_trader*, which will be explained later on. Otherwise, if the initial penalty has been successfully decreased, each penalty  $z_i$  is updated according to the new schedule, and *penalty\_decreaser* starts again its execution. When no further penalty reduction is obtained, the *greedy\_scheduler* is restarted. We remark that, in this attempt to decrease the penalty of the timetable, *penalty\_decreaser* could even find a timetable with fewer time slots. If this happens,  $T$  is updated accordingly.

When both penalties and number of time slots  $T$  can not be changed by the *greedy\_scheduler* and the *penalty\_decreaser*, we invoke the *penalty\_trader*, which tries to trade off penalties for time slots. In other words, we check whether the penalty of the current schedule can be decreased by allocating one additional time slot (i.e.,  $T + 1$ ). To do this, we have to choose which exams should be moved from their current time slot to  $(T + 1)$ . We define  $\Delta_i$  as follows:  $\Delta_i = \left( \sum_{j \in E: (i,j) \in \mathcal{E}} |(T + 1) - t_j| \right) - \left( \sum_{j \in E: (i,j) \in \mathcal{E}} |t_i - t_j| \right)$ . The intuition behind  $\Delta_i$

is the following. Note that  $\left(\sum_{j \in E: (i,j) \in \mathcal{E}} |t_i - t_j|\right)$  gives the distance, measured in time slots, between exam  $i$  and its neighbors when  $i$  is assigned to time slot  $t_i$ , and  $\left(\sum_{j \in E: (i,j) \in \mathcal{E}} |(T+1) - t_j|\right)$  gives the same quantity when  $i$  is assigned to time slot  $T+1$ . Thus,  $\Delta_i$  measures the difference between those two distances: as the penalty is non-decreasing with the time slot difference,  $\Delta_i > 0$  indicates that the overall penalty can be decreased by moving exam  $i$  from time slot  $t_i$  to time slot  $T+1$ . Similarly,  $\Delta_i < 0$  indicates that the overall penalty is increased when exam  $i$  is moved from time slot  $t_i$  to time slot  $T+1$ . The *penalty-trader* neglects exams with  $\Delta_i < 0$  and sorts the remaining exams by non-decreasing values of  $\Delta_i$ . It then considers exams in this ordered list: it assigns exam  $i$  to time slot  $T+1$  and deletes its neighbors from the list. This continues until the (stable) set formed by the examinations moved to time slot  $(T+1)$  is maximal. Whenever there are no examinations  $i$  with  $\Delta_i \geq 0$ , the *penalty-trader* can not make any penalty improvement: in this case, it repeats the whole process with increasing values of  $T$ , say  $T+2$ ,  $T+3$ ,  $\dots$ . When the *penalty-trader* finds a larger  $T$  for which the penalty is decreased, the priorities are reassigned as  $p_i = 1/t_i$  and the *greedy-scheduler* is invoked again.

We note that our approach is based on the generation of similar solutions: i.e., we try to perturb the current solution with a certain number of time slots, in the attempt of spacing out the examinations more evenly. This has also some drawbacks, however, as the algorithm can get stuck for a long time in particular areas of the solution space. To avoid this, we use a simple but effective *checkpointing* scheme: the algorithm stops at certain steps, releases basically all of its memory, and starts a new local search after this. We now analyze checkpointing in detail. Define an *iteration* as an attempt (not necessarily successful) to change the current state of an examination  $i$  (i.e., change either its time slot  $t_i$  or its penalty  $z_i$ ). Roughly speaking, we force a checkpoint after a certain number of iterations: after each checkpoint, our algorithm restarts again a new local search from the current timetable, with a consequent release of memory used. Releasing memory once in a while helps in decreasing the space consumption. This has a positive effect on the running times as well, as algorithms with high space usage tend to be very slow in practice. Moreover, we noticed in our experiments that a properly tuned checkpoint is beneficial on large instances: indeed, it prevents that the portion of the conflict graph examined by local search becomes too large, and forces the algorithm to work on smaller portions of the graph. We implemented two different types of checkpointing: *constant* and *adaptive*. The first is the simpler: the interval at which checkpointing is performed is constant, i.e., checkpointing is performed exactly every  $\ell$  iterations, with  $\ell$  being a properly chosen constant. The second is a bit more complicated, as the checkpointing interval depends on the number of iterations performed, i.e., it grows as the algorithm performs more iterations. Namely, let  $\ell_0$  be the initial checkpointing interval, and  $\alpha > 1$  be a constant. Let  $\ell_i = \alpha \ell_{i-1}$ ,  $i \geq 1$ . For the first  $\ell_1$  iterations (i.e., in the range  $(0, \ell_1]$ ), the algorithm performs checkpointing at interval  $\ell_0$ . For the subsequent  $(\ell_2 - \ell_1)$  iterations (i.e., in the range  $(\ell_1, \ell_2]$ ), the algorithm performs checkpointing at interval  $\ell_1 > \ell_0$ . In general, for iterations in the range

$(\ell_i, \ell_{i+1}]$ , the algorithm performs checkpointing at interval  $\ell_i > \ell_{i-1}$ . We will analyze in details the differences between these two checkpointing schemes in Section 3.

Unfortunately, the checkpointing scheme can introduce also some drawbacks. Namely, when we release all the memory used due to checkpointing, we are also risking to loose some important information, such as solutions previously explored, or data which could have been helpful in avoiding getting trapped in a local minimum. To solve this problem, we introduce a careful assignment of priorities after a checkpointing, which we call *bridging priorities*. We do this follows. Define a *phase* as the interval between any two successive checkpoints. As the algorithm retains no information of what happened before a checkpoint, it could explore solutions which are exactly the same or very similar to solutions that were generated in previous phases, without being able to notice it. To circumvent this problem the algorithm relies on an appropriate reassignments of priorities to the examinations after each checkpointing: this dynamic reassignment of priorities tries to bridge subsequent phases by bypassing the checkpoints.

The integration of these two features, i.e., checkpointing and bridging priorities, allows one to start successive local searches which are closely interacting, without increasing the space consumption. We now define how priorities are re-assigned. Define the *update count* of a vertex  $v$  as the number of times  $v$  changed its state in the current phase (i.e., since the last checkpoint). Roughly speaking, the update count of an examination measures its active participation during that phase. After a checkpoint and before the update counts are reset, the examination priorities are set as  $p_i = 1/\text{update\_count}$ . With this assignment, a vertex with a low update count (i.e., which was not very active in the last phase) gets a high priority, and thus there is a better chance to get this vertex involved in the next phases. This clearly prevents the algorithm to continue its visit on a same subset of vertices. In our experiments, this particular choice of priorities generated a sequence of phases which were interacting closely with each other: in most cases a phase was building on the priorities and solutions offered by the previous phase, and consistently offered better priorities and solutions to the next phase. We implemented and tested four variants of our algorithm, depending on the type of checkpointing used, and on whether bridging priorities were used or not:

- ACP: uses Adaptive Checkpointing and bridging Priorities;
- AC: uses Adaptive Checkpointing but not bridging priorities;
- CCP: uses Constant Checkpointing and bridging Priorities;
- CC: uses Constant Checkpointing but no bridging priorities.

As a result of our engineering efforts, we decided to combine ACP and CC into a hybrid algorithm, called ACCP. ACCP works in two steps. In the first step, it tries to obtain a conflict-free solution with as few time slots as possible. This is accomplished by running ACP on the input instance. Let  $T$  be the number of time slots obtained. Next, ACCP tries to reduce the penalty of the solution while keeping at  $T$  the number of time slots. This is done by running a modified version of CC on the initial input instance. In this version of CC, we did not use either the *greedy\_scheduler* or the *penalty\_trader*, as there was no need to change (i.e.,

either decrease or increase) the number of time slots in the current solution. Instead of these two functionalities, we used a simpler *slot\_swapper*, which in some sense accomplished the same task as the *penalty\_trader*, without moving to a higher number of time slots: the *slot\_swapper* was executed whenever either the *greedy\_scheduler* or the *penalty\_trader* were called before. More in details, the *slot\_swapper* swaps examinations in time slot  $t_i$  with examinations in time slots  $t_j$ , for  $t_i \neq t_j$ , as follows. Define the quantity  $\Delta_{ij}$  for each  $1 \leq i, j \leq T$ ,  $i \neq j$ , as  $\Delta_{ij} = \sum_{h \in E: t_h = i} \sum_{k \in E: (h,k) \in \mathcal{E}} |t_j - t_k| + \sum_{h \in E: t_h = j} \sum_{k \in E: (h,k) \in \mathcal{E}} |t_i - t_k| - \sum_{h \in E: t_h = i} \sum_{k \in E: (h,k) \in \mathcal{E}} |t_i - t_k| - \sum_{h \in E: t_h = j} \sum_{k \in E: (h,k) \in \mathcal{E}} |t_j - t_k|$ . Intuitively,  $\Delta_{ij}$  relates to the difference in solutions given by swapping exams in time slot  $t_i$  with exams in time slot  $t_j$ . We take  $i$  and  $j$  maximizing  $\Delta_{ij}$  and swap all exams currently assigned to time slot  $t_i$  with all the exams currently assigned  $t_j$ . If  $\max_{i,j} \{\Delta_{ij}\} < 0$  the algorithm terminates. After the *slot\_swapper* has run, the *penalty\_decreaser* is invoked and the algorithm keeps on iterating between these two functionalities until the stopping criterion is met.

As it was already mentioned in the introduction, in many timetabling applications one is concerned with a fixed number of time slots, and thus wishes to space out as evenly as possible (according to the proximity constraints) a given number of exams within these time slots. This can be easily done by feeding ACCP with the desired number of time slots. In this special case, the underlying call to ACP accomplishes a simpler task. Indeed, rather than spending a lot of efforts in generating a solution with the lowest possible number of time slots, it can just provide the first solution which meets the number of time slots given in input.

### 3 Experimental Results

**Data Sets.** Benchmarks for examination timetabling are publicly available at [ftp.cs.nott.ac.uk/ttp/Data](http://ftp.cs.nott.ac.uk/ttp/Data) and at [ie.utoronto.ca/pub/carter/testprob](http://ie.utoronto.ca/pub/carter/testprob). The data sets consist of real-life examples taken from several Canadian Institutions, from the London School of Economics, from King Fahd University of Petroleum, Minerals, Dhahran, from Purdue University, and from the Nottingham University. Each data set is identified by a code, where the first three letters refer to the Institution, *F* or *S* stand for the semester (Fall or Spring), and the last two digits to the year. For instance, PUR-S-93 refers to the examinations scheduled in the Spring '93 at Purdue University. In these benchmarks, the number of students ranges from 611 (STA-F-83) to 30,032 (PUR-S-93), and the number of examinations to be scheduled ranges from 81 (HEC-S-92) to 2,419 (PUR-S-93).

**Experimental Setup.** Our algorithms have been coded in C and the experiments run on a PC with a 500MHz Pentium and 64MB of RAM. For lack of space we report here only the results of two sets of experiments. The first considers only proximity constraints (and their associated penalties) but no side constraints. The results are illustrated in Tables 1, 2 and 3. The second set of experiments

considers also room capacity constraints: i.e., a maximum number of available seats per time slot is given, and thus only a limited number of students can be examined in each time slot. Results of this test are reported in Table 4. In each experiment, we collected data about the number of time slots found by a given algorithm, its associated penalty (normalized on the total number of students) and its running times in seconds. The normalized penalty allows one to compare results on instances of different size.

We compared our algorithms with the five algorithms in [7], denoted by **Carter**, the algorithms in [2], denoted by **Burke** and the algorithms in [9], denoted by **Di Gaspero**. We had no access to the codes of Burke *et al.*, Carter *et al.* and Di Gaspero and Schaerf, and thus the data related to **Burke**, **Carter** and **Di Gaspero** are taken verbatim from their papers [2,7,9]. This implies that running times are not necessary meaningful, as they relate to different platforms. In order to ensure a fair comparison, we tried to reproduce for our algorithms exactly the same experimental setup reported in [2,7,9]: in particular, like the other implementations, we stopped the tests for our algorithms whenever they were not able to improve the solutions found after a certain number of iterations. We remark that other stopping criteria can be easily implemented in our code.

**Experimental Analysis.** Table 1 lists the results of the same experiments reported on Table 3 of [7]. We start by analyzing the relative behavior of **ACP**, **AC**, **CCP** and **CC** in this set of experiments. Next, we comment on **ACCP**. Finally, we summarize the performance of our algorithms with respect to the algorithms of Carter *et al.* [7].

The first feature that can be observed from columns **CC**, **CCP**, **AC** and **ACP** of Table 1 is that algorithms that produce better solutions (i.e., solutions with a smaller number of time slots) have consistently higher penalties. This seems natural, as solutions with a higher number of time slots have more room to space out examinations evenly. Secondly, it can be easily observed that adaptive checkpointing outperforms consistently constant checkpointing, i.e., the solutions produced by **ACP** (respectively **AC**) have fewer number of time slots than the solutions produced by **CCP** (respectively **CC**). We explain this phenomenon as follows. In the initial stages, adaptive checkpointing tends to create a larger number of phases than constant checkpointing, which could be helpful in exploring the solution space. Indeed, we noticed from a profiling that adaptive checkpointing issued a larger number of calls to the *greedy\_scheduler* in the initial phases. This made adaptive checkpointing more prone to finding a good solution (with a small number of time slots) before constant checkpointing did. As the algorithm progresses, adaptive checkpointing makes the number of phases decrease significantly (with respect to constant checkpointing): this implies that, as time goes by, adaptive checkpointing works more with the *penalty\_decreaser* and the *penalty\_trader* rather than with the *greedy\_scheduler*, and thus decreases the penalty rather than the number of time slots. The same effect can be noticed for bridging priorities, as the solutions produced by **ACP** (respectively **CCP**) have consistently fewer number of time slots than the solutions produced by **AC** (respectively **CC**). It thus seems that tunneling information through subsequent

**Table 1.** Comparison with the results of Carter *et al.* [7].

Data_set		CC	CCP	AC	ACP	ACCP	Carter LD	Carter SD	Carter LWD	Carter LE	Carter RO
CAR-F-92	Slots	32	30	29	<b>28</b>	<b>28</b>	31	<b>28</b>	30	31	32
	Penalty	<b>6.0</b>	6.4	6.8	7.0	6.4	—	—	—	—	—
	Time	428.9	398.2	475.9	486.7	559.2	618.3	510.8	911.6	121.2	1143.3
CAR-S-91	Slots	32	32	30	<b>28</b>	<b>28</b>	32	<b>28</b>	30	32	35
	Penalty	<b>6.8</b>	7.0	7.2	7.8	7.3	—	—	—	—	—
	Time	82.9	58.4	60.8	70.7	86.3	293.8	75.1	1046.4	210.4	712.7
EAR-F-83	Slots	23	23	23	<b>22</b>	<b>22</b>	23	24	23	23	24
	Penalty	<b>40.2</b>	40.8	42.0	46.8	43.2	—	—	—	—	—
	Time	140.8	104.9	99.6	111.8	150.1	80.2	173.0	156.6	151.8	256.5
HEC-S-92	Slots	18	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	18	<b>17</b>	<b>17</b>
	Penalty	<b>9.2</b>	10.8	10.8	11.2	10.6	—	—	—	—	—
	Time	24.2	17.8	12.8	10.2	19.9	31.8	17.1	22.8	219.5	83.1
KFU-S-93	Slots	20	<b>19</b>	<b>19</b>	<b>19</b>	<b>19</b>	20	<b>19</b>	<b>19</b>	<b>19</b>	<b>19</b>
	Penalty	<b>13.8</b>	14.2	14.2	15.9	14.0	—	—	—	—	—
	Time	128.2	110.6	110.4	85.9	159.6	120.1	97.2	2396.5	1852.7	2217.0
LSE-F-91	Slots	18	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>
	Penalty	<b>9.6</b>	9.8	10.8	13.4	<b>9.6</b>	—	—	—	—	—
	Time	195.8	178.1	145.7	122.8	223.4	78.7	164.6	763.9	786.2	674.5
PUR-S-93	Slots	42	40	38	<b>36</b>	<b>36</b>	43	38	40	40	42
	Penalty	<b>3.7</b>	3.9	4.1	4.8	4.4	—	—	—	—	—
	Time	5874.8	4289.4	4256.0	4564.0	6021.2	31922.2	73338.3	8782.3	9304.9	72028.5
RYE-F-93	Slots	23	22	<b>21</b>	<b>21</b>	<b>21</b>	23	<b>21</b>	22	22	22
	Penalty	<b>6.8</b>	7.4	7.8	8.0	7.7	—	—	—	—	—
	Time	125.7	102.2	98.4	142.5	225.9	507.2	380.0	419.4	502.5	3467.4
STA-F-83	Slots	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>
	Penalty	158.2	162.0	168.4	173.8	<b>150.2</b>	—	—	—	—	—
	Time	9.2	5.7	5.5	4.0	10.2	6.0	6.0	5.7	6.1	98.9
TRE-S-92	Slots	23	22	21	<b>20</b>	<b>20</b>	22	21	21	23	22
	Penalty	<b>9.4</b>	9.8	10.2	10.8	10.3	—	—	—	—	—
	Time	115.7	95.2	94.5	100.1	214.7	398.8	108.4	1067.9	107.4	836.3
UTA-S-93	Slots	34	32	31	<b>30</b>	<b>30</b>	33	35	35	34	35
	Penalty	<b>3.5</b>	4.2	4.5	5.0	4.4	—	—	—	—	—
	Time	760.5	698.2	685.9	758.2	1023.5	4294.1	2557.2	9223.0	7340.4	17323.1
UTE-S-92	Slots	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>
	Penalty	28.4	32.2	38.5	41.2	<b>24.3</b>	—	—	—	—	—
	Time	7.2	6.8	5.8	4.2	9.8	6.3	4.7	8.5	9.1	9.9
YOR-F-83	Slots	21	20	20	<b>19</b>	<b>19</b>	21	<b>19</b>	20	20	20
	Penalty	<b>36.2</b>	42.2	44.0	45.9	44.2	—	—	—	—	—
	Time	138.7	134.9	120.2	148.2	226.2	649.4	190.4	67.1	516.4	351.5

phases can improve substantially the performance of the *greedy\_scheduler*. In summary, among the four implementations, ACP seems to give always the best solution in terms of number of time slots, while CC produces solutions with the highest number of time slots but lowest penalties. The performance of AC and CCP is within the boundaries defined by ACP and CC, with AC producing consistently fewer time slots and higher penalties than CCP. One could thus use ACP if a solution with minimum schedule length is desired or alternatively use CC if time slots are not an issue and a timetabling with low penalty is preferred. Schedules obtained with AC and CCP represent other tradeoffs between these two objectives, with emphasis on time slots for AC and on penalties for CCP. We now comment on the solutions found by ACCP. As expected, the number of time slots found by ACCP coincide with the time slots of ACP, with ACCP having higher running times than ACP (as ACCP runs ACP as a subroutine). As shown in Table 1, the extra work done by ACCP seems extremely useful in reducing the solution penalties. It

**Table 2.** ACP, AC, CCP, CC, and ACCP executed with a fixed CPU time.

Data_set		CC	CCP	AC	ACP	ACCP
CAR-F-92	Slots	32	30	29	<b>28</b>	<b>28</b>
	Penalty	<b>6.0</b>	6.2	6.8	7.0	6.4
	Time	1120	1120	1120	1120	1120
CAR-S-91	Slots	32	31	30	<b>28</b>	<b>28</b>
	Penalty	<b>6.8</b>	7.0	7.2	7.6	7.2
	Time	172	172	172	172	172
EAR-F-83	Slots	23	23	23	<b>22</b>	<b>22</b>
	Penalty	<b>40.0</b>	40.4	41.5	46.4	43.0
	Time	300	300	300	300	300
HEC-S-92	Slots	18	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>
	Penalty	<b>9.2</b>	10.6	10.7	11.0	10.5
	Time	48	48	48	48	48
KFU-S-93	Slots	20	<b>19</b>	<b>19</b>	<b>19</b>	<b>19</b>
	Penalty	<b>13.6</b>	14.1	14.2	15.3	14.0
	Time	320	320	320	320	320
LSE-F-91	Slots	18	<b>17</b>	<b>17</b>	<b>17</b>	<b>17</b>
	Penalty	<b>9.6</b>	9.8	10.5	12.9	<b>9.6</b>
	Time	446	446	446	446	446
PUR-S-93	Slots	42	40	38	<b>36</b>	<b>36</b>
	Penalty	<b>3.6</b>	3.8	4.0	4.7	4.4
	Time	12042	12042	12042	12042	12042
RYE-F-93	Slots	23	22	<b>21</b>	<b>21</b>	<b>21</b>
	Penalty	<b>6.8</b>	7.3	7.7	8.0	7.7
	Time	452	452	452	452	452
STA-F-83	Slots	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>	<b>13</b>
	Penalty	158.2	162.0	168.2	173.4	<b>150.2</b>
	Time	30	30	30	30	30
TRE-S-92	Slots	23	22	21	<b>20</b>	<b>20</b>
	Penalty	<b>9.4</b>	9.7	10.0	10.8	10.3
	Time	430	430	430	430	430
UTA-S-93	Slots	34	32	31	<b>30</b>	<b>30</b>
	Penalty	<b>3.5</b>	4.1	4.4	5.0	4.4
	Time	2046	2046	2046	2046	2046
UTE-S-92	Slots	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>	<b>10</b>
	Penalty	28.4	32.2	36.6	40.0	<b>24.3</b>
	Time	20	20	20	20	20
YOR-F-83	Slots	21	20	20	<b>19</b>	<b>19</b>
	Penalty	<b>36.0</b>	42.0	43.8	45.6	44.0
	Time	452	452	452	452	452

was somehow surprising for us to notice that often the penalties of ACCP were lower than those reported by AC or CCP (even when AC and CCP produced a higher number of time slots). Furthermore, in the cases where the solutions found by ACP, AC, CCP and CC had the same number of time slots (such as STA-F-83 and UTE-S-92), ACCP was even able to improve on the penalty achieved by CC. One could wonder whether the stopping criterion adopted in this experiment (i.e., stopping whenever a given algorithm is able to improve the solutions found after a certain number of iterations) might give an unfair advantage to CC and ACCP, which tend to have higher running times than CCP, AC and ACP. This does not seem to be the case. Indeed, we performed another experiment on the same data sets, in which we ran each algorithm for exactly the same (CPU) time. As it can be noted from Table 2, which reports the results of this experiment, time slots and penalties are very similar to those obtained in the Table 1, except only for slight improvements on penalties.

**Table 3.** Comparison with the results of Carter *et al.* [7] and of Di Gaspero and Schaerf [9].

Data_set		ACCP	Carter LD	Carter SD	Carter LWD	Carter LE	Carter RO	Di Gaspero
CAR-F-92	Slots	32	32	32	32	32	32	—
	Penalty	<b>6.0</b>	7.6	6.6	6.6	6.2	8.2	—
	Time	142.7	637.9	15.0	886.0	47.0	1143.4	—
CAR-S-91	Slots	35	35	35	35	35	35	—
	Penalty	<b>6.6</b>	7.9	7.1	7.4	7.6	11.9	—
	Time	34.7	29.2	20.7	67.5	58.3	712.7	—
EAR-F-83	Slots	24	24	24	24	24	24	24
	Penalty	<b>29.3</b>	36.4	46.5	37.3	42.3	42.1	49.4
	Time	100.2	24.7	6.0	76.1	242.4	256.5	40.8
HEC-S-92	Slots	18	18	18	18	18	18	18
	Penalty	<b>9.2</b>	10.8	12.7	15.8	15.9	20.0	15.4
	Time	11.0	7.4	6.5	22.8	8.7	54.9	13.4
KFU-S-93	Slots	20	20	20	20	20	20	20
	Penalty	<b>13.8</b>	14.0	15.9	22.1	20.8	21.0	19.7
	Time	112.8	120.2	559.4	147.3	208.9	977.6	57.2
LSE-F-91	Slots	18	18	18	18	18	18	18
	Penalty	<b>9.6</b>	12.0	12.9	13.1	10.5	13.3	16.2
	Time	92.8	50.3	50.5	233.5	48.0	674.5	29.0
PUR-S-93	Slots	42	42	42	42	42	42	—
	Penalty	<b>3.7</b>	4.4	4.1	5.0	3.9	—	—
	Time	4018.3	31922.1	22921.9	10601.1	21729.4	>200000	—
RYE-F-93	Slots	23	23	23	23	23	23	—
	Penalty	<b>6.8</b>	7.3	7.4	10.0	7.7	11.0	—
	Time	89.4	507.2	403.0	360.0	391.9	3019.2	—
STA-F-83	Slots	13	13	13	13	13	13	13
	Penalty	<b>158.2</b>	162.9	165.7	161.5	161.5	184.1	159.8
	Time	6.5	6.0	6.0	5.7	6.1	98.9	10.1
TRE-S-92	Slots	23	23	23	23	23	23	23
	Penalty	<b>9.4</b>	11.0	10.4	9.9	9.6	13.0	10.8
	Time	102.8	150.6	89.3	74.1	107.4	468.4	137.4
UTA-S-93	Slots	35	35	35	35	35	35	35
	Penalty	<b>3.5</b>	4.5	3.5	5.3	4.3	6.4	5.0
	Time	589.4	1287.8	664.3	5929.7	2705.1	7135.6	1351.0
UTE-S-92	Slots	10	10	10	10	10	10	10
	Penalty	<b>24.4</b>	38.3	31.5	26.7	25.8	31.6	30.6
	Time	5.0	6.3	4.7	8.5	9.1	9.9	6.9
YOR-F-83	Slots	21	21	21	21	21	21	21
	Penalty	<b>36.2</b>	49.9	44.8	41.7	45.1	46.5	46.2
	Time	125.4	215.7	428.0	271.4	174.5	231.8	32.0

Another feature that comes out from Tables 1 and 2 is that in all tested benchmark instances ACCP consistently achieves the lowest number of time slots, and that its penalty is the lowest among all the algorithms achieving exactly this lowest number of time slots.

We now go back to Table 1 and compare our algorithms to the algorithms of [7]. As it can be easily seen, all our algorithms, and in particular ACP and ACCP, always find consistently solutions with fewer time slots than the algorithms of Carter *et al.* Note that Carter *et al.* did not report the penalties of the solutions found by their algorithms in this experiment. To give an idea about this, they set up another experiment: for each benchmark instance they selected a fixed number of time slots (taken as the maximum number of time slots found by their algorithms in the previous experiment), and reported the penalties found by their algorithms with this fixed number of time slots. The results of this test are summarized in column **Carter** of Table 3.



**Table 4.** Comparison with the results of Burke *et al.* [2] and of Di Gaspero and Schaerf [9].

Data_set		ACCP	Burke	Di Gaspero
CAR-F-92	Slots	40	40	40
	Penalty	<b>268</b>	331	443
	Time	80.4	–	–
CAR-S-91	Slots	51	51	51
	Penalty	<b>74</b>	81	98
	Time	31.4	–	–
KFU-S-93	Slots	20	20	20
	Penalty	912	974	<b>597</b>
	Time	118.2	–	–
TRE-S-92	Slots	35	35	35
	Penalty	<b>2</b>	3	5
	Time	222.4	–	–
NOTT	Slots	26	26	26
	Penalty	44	53	<b>13</b>
	Time	359.1	–	–
UTA-S-93	Slots	38	38	38
	Penalty	680	772	<b>625</b>
	Time	265.1	–	–

The same experiment was performed by Di Gaspero and Schaerf as well, at least for a subset of benchmark instances: the results are reported on the last column of Table 3 (averaged on several executions of their algorithms). In order to make a fair comparison, we organized for ACCP the same experimental setup, and the results are illustrated in column ACCP. As it can be easily seen, the penalties obtained by ACCP are always substantially lower than the range of penalties achieved either by the algorithms of Carter *et al.* or by the algorithms of Di Gaspero and Schaerf. Moreover, our algorithms look more robust and less sensitive to the problem instance. Namely, CC, CCP, AC, ACP and ACCP seem to maintain the same relative behavior on all input instances: e.g., ACCP and ACP always find the solutions with fewer time slots, and CC the solutions with smaller penalties. The algorithms of Carter *et al.* and by Di Gaspero and Schaerf, on the contrary, seem quite sensible to the problem instance, and algorithms of the same family show quite different behaviors on different input instances. This might be of practical interest, as having to select one among several programs based on the input instance, or having to tune the program parameters as a function of the input data might reveal to be either too cumbersome or impractical.

Finally, Table 4 contains some results on experiments with room capacity constraints on a fixed number of time slots. We report the data obtained from the same experiment set up by Burke *et al.* [2] and reproduced by Di Gaspero and Schaerf: column ACCP contains information about the solutions achieved by ACCP and columns Burke and Di Gaspero contain the data reported in [2] and in [9], respectively. To keep consistency with [2] and [9], differently from Tables 1–3, in Table 4 we report the overall penalties found by the algorithms (i.e., penalties are not normalized over the total number of students). The table shows that ACCP yields a substantial improvement over the solution quality of Burke, and is superior to Di Gaspero in many cases.

## Acknowledgments

We are indebted to the anonymous referees for their comments on a previous version of this paper. We are also grateful to Andrea Schaerf for sending us a copy of [9].

## References

1. Burke, E., and Newall, J., "A Multi-Stage Evolutionary Algorithm for the Timetabling Problem", *IEEE Transactions on Evolutionary Computation*, 3 (1) (1999) 63–74.
2. Burke, E., Newall, J., and Weare, R., "A Memetic Algorithm for University Exam Timetabling", in *Proc. of the 1st Int. Conf. on the Practice and Theory of Automated Timetabling*, (1995) 241–250.
3. Corne, D., Fang, H.-L., and Mellish, C., "Solving the Modular Exam Scheduling Problem with Genetic Algorithms", Tech. rep. 622, Department of Artificial Intelligence, University of Edinburgh, (1993).
4. Corne, D., Ross, P., and Fang, H.-L., "Evolutionary Timetabling: Practice Prospects and Work in Progress", in *UK Planning and Scheduling SIG Workshop*, (1994).
5. Carter, M.W., "A Survey of Practical Applications of Examination Timetabling Algorithms", *Operations Research*, 34 (2) (1986) 193–202.
6. Carter, M.W., Laporte, G., and Chinnek, J.W., "A General Examination Scheduling", *Interfaces*, 24 (3) (1994) 109–120.
7. Carter, M.W., Laporte, G., and Lee, S.Y., "Examination Timetabling: Algorithm Strategies and Applications", *Journal of the Operational Research Society*, 47 (1996) 373–383.
8. De Werra, D., "An Introduction to Timetabling", *European Journal of Operational Research*, 19 (1985) 151–162.
9. Di Gaspero, L., and Schaerf, A., "Tabu Search Techniques for Examination Timetabling Problems", in *Proc. of the 6th Int. Conf. on the Practice and Theory of Automated Timetabling*, (2000).
10. Garey, M.R., and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-completeness*, W.H. Freeman & Company, San Francisco, (1979).
11. Johnson, D., "Timetabling University Examinations", *Journal of the Operational Research Society*, 41 (1990) 39–47.
12. Laporte, G., and Desroches, S., "Examination Timetabling by Computer", *Computers and Operations Research*, 11 (1984) 351–360.
13. Metha, N.K., "The Application of a Graph Colouring Method to an Examination Scheduling Problem", *Interfaces*, 11 (5) (1981) 57–64.
14. Thompson, J.M., and Dowsland, K.A., "A Robust Simulated Annealing Based Examination Timetabling System", *Computers and Operations Research*, 25 (1998) 637–648.
15. Schaerf, A., "A Survey of Automated Timetabling", *Artificial Intelligence Review*, 13 (2) (1999) 87–127.

## Author Index

- Albers, Susanne 11  
Arge, Lars 51  
Aslam, Javed 74  
  
Backes, Werner 63  
Bojesen, Jesper 159  
  
Caramia, Massimiliano 230  
Chase, Jeff 51  
Chatzigiannakis, I. 99  
  
Dell’Olmo, Paolo 230  
Demetrescu, Camil 147, 218  
  
Edelkamp, Stefan 39  
Erlebach, Thomas 195  
  
Finocchi, Irene 147  
Fleischer, Rudolf 135  
Frigioni, Daniele 218  
  
Guérin Lassous, Isabelle 111  
Gustedt, Jens 111  
  
Halperin, Dan 171  
Hanniel, Iddo 171  
Hinrichs, Klaus H. 183  
  
Italiano, Giuseppe F. 230  
Iwama, Kazuo 123  
  
Jansen, Klaus 195  
  
Katajainen, Jyrki 159  
Kawai, Daisuke 123  
  
Leblanc, Alain 74  
Liebers, Annegret 87  
Liotta, Giuseppe 147  
  
Marchetti-Spaccamela, Alberto 218  
Mehlhorn, Kurt 23  
Miyazaki, Shuichi 123  
  
Nanni, Umberto 218  
Nikoletseas, S. 99  
  
Okabe, Yasuo 123  
  
Proietti, Guido 207  
  
Sanders, Peter 135  
Schäfer, Guido 23  
Schröder, Bianca 11  
Stein, Clifford 74  
Spirakis, P. 99  
Stiegeler, Patrick 39  
  
Umemoto, Jun 123  
  
Vahrenhold, Jan 183  
Vitter, Jeffrey S. 51  
  
Weihe, Karsten 1, 87  
Wetzel, Susanne 63  
Wickremesinghe, Rajiv 51